UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

UN EDITOR GRÁFICO PARA EL DISEÑO POR CONTRATO EN ECLIPSE

Víctor López Rivero

Enero 2014

Resumen

El proyecto que se presenta se titula “Un editor gráfico para el diseño por contrato en Eclipse” y por lo tanto lo que se ha realizado es un editor gráfico para definir gráficamente contratos para transformaciones de modelos, siendo estas últimas, programas que reciben un modelo de entrada y generan un modelo de salida. Estas transformaciones se utilizan intensivamente en el paradigma de desarrollo de software dirigido por modelos.

Todo comienza cuando una serie de desarrolladores crean un lenguaje para la ingeniería de transformaciones de modelos. Además de este lenguaje, también se crea un editor para definir esos contratos textualmente.

Frente al editor textual ya existente surge la idea de por qué no crear un editor gráfico sobre ese lenguaje para la ingeniería de transformaciones de modelos. Entre todas las herramientas existentes para poder llevar esta idea a cabo, se escoge Graphiti, un plug-in de Eclipse mediante el cual cualquier usuario que tenga esta última herramienta puede generar editores gráficos de apariencia profesional de una manera rápida y sencilla.

Este editor gráfico dispone de un lienzo y de una paleta de la cual puedes arrastrar elementos al lienzo y éstos queden ahí plasmados. El usuario puede mover, redimensionar o eliminar todo a su gusto para que la creación de diagramas sea fluida y funcional.

El trasfondo de este proyecto es el de poder definir el comportamiento esperado de las transformaciones de modelos gráficamente (seguir una traza de transformación) por lo que es posible que en un futuro sea posible validar estas transformaciones gráficamente ampliando este proyecto.

Palabras clave: diseño por contrato, entornos gráficos de modelado de software, Graphiti, transformación de modelos, desarrollo dirigido por modelos.

Abstract

The presented project is called “A graphical editor for contract design in Eclipse” therefore it has been created an editor to graphically define model transformation contracts, being the latter programs which receive an input model and generate an output model. Model transformations are used intensively in the paradigm of model-driven software development.

Everything begins when some developers create a modelling language to engineer model transformations. Besides this language, a textual editor to define contracts for model transformations is also created.

To complement the existing textual editor, it arises the idea of creating a graphical editor for this contract-based language. Among all existing tools that can realise this idea, Graphiti is chosen. Graphiti is an Eclipse framework which allows the generation of professional looking graphical editors quickly and easily.

This graphical editor has a canvas and a palette in which one can drag-and-drop elements to the canvas and so they are created. The user can move, resize or delete everything he/she pleases to create the diagram in a quick and functional way.

The background of this project is to define the expected behaviour of model transformations graphically (keep a trace of the transformation) so that in a near future the project can be expanded with support to validate transformations graphically.

Key words: design by contract, graphical enviroments for software modelling, Graphiti, model transformations, model driven engineering.

Agradecimientos

En primer lugar y lo más importante, dar las gracias a Esther por todo lo que me ha ayudado y por todas esas veces que ha tenido que dejar de hacer algo para contestar a la lista interminable de correos que le enviaba.

También doy gracias a esas personas que me han acompañado durante la carrera. Ellos sabrán si incluirse en este párrafo o no. No pensaba yo que fuera posible reírse tanto estudiando cosas serias pero vaya, ya os digo yo que sí. Gracias a los profesores no tan buenos que he tenido que también me han enseñado cosas. Cómo no, gracias a los grandes profesores que han hecho que de verdad aprenda y disfrute al mismo tiempo y que saben cómo tratar a los alumnos.

Y...

...por último, gracias a mis padres y a mis amigos que se han encargado de que mi cerebro desconecte de vez en cuando. Faltaría más, no todo va a ser trabajar.

Índice de contenido

Índice de figuras

1. Introducción

1. 1. Antecedentes

La ingeniería dirigida por modelos (Model Driven Engineering - MDE [1]) es un paradigma de desarrollo de software que eleva el nivel de abstracción y automatización en la construcción de software mediante la utilización de modelos. A partir de un modelo de entrada se obtiene el producto final. Éste puede ser otro modelo e incluso un código autogenerado a partir de la entrada. Está orientado para trabajar a alto nivel (lo cual facilita las cosas) y a partir de un solo modelo es posible conseguir varios artefactos al mismo tiempo, u obtener diversos productos para distintas plataformas.

Como hemos dicho, a partir del modelo de entrada se pueden obtener diferentes productos finales. En nuestro caso nos centraremos en los casos en los que la salida es otro modelo. A esto lo llamamos una transformación modelo a modelo (M2M) y para construirlas existen unos lenguajes de programación específicos. Algunos ejemplos de estos lenguajes de transformación M2M son ATL [2] y QVT [3] entre otros. Básicamente estos lenguajes contemplan dos modelos (uno de entrada y otro de salida) y establecen una serie de reglas que relacionan qué elemento de la entrada se va a convertir en qué elemento de la salida. También puede haber casos menos sencillos y, por ejemplo, en lugar de tener dos modelos tenemos más (puede haber un estado intermedio de transformación). El fin de estas transformaciones varía pero en algunas ocasiones se realizan para pasar de un modelo independiente de la plataforma a un modelo específico de la plataforma, para simulación, para obtener un refinamiento del modelo original, para traducir un modelo a otro lenguaje donde hacer análisis de interés, etc.

transML [4] es una familia de lenguajes para la ingeniería de transformaciones de modelos, desarrollado en el grupo miso de la Universidad Autónoma de Madrid. Entre otros, incluye un lenguaje para la especificación de contratos para transformaciones de modelos llamado Pamomo. El lenguaje está definido mediante un metamodelo, y en la actualidad existe un editor para definir dichos contratos textualmente, que además automatiza la generación de modelos de prueba y aserciones que facilitan la fase de pruebas de la transformación implementada. Brevemente, este lenguaje permite expresar invariantes que establecen relaciones entre los modelos de entrada y los de salida de una transformación. Cada invariante puede definir precondiciones positivas y negativas, que definen condiciones requeridas o prohibidas para el cumplimiento de la cláusula del contrato (estando cada cláusula definida mediante un patrón). De este modo, una cláusula del contrato debe cumplirse para todas las ocurrencias del grafo origen que cumplen las precondiciones positivas y no cumplen las negativas.

Como bien hemos explicado en la sección anterior, el objetivo del proyecto es desarrollar una herramienta que nos permita manipular contratos para los programas de transformación de modelos integrada en Eclipse [5]. Puesto que no existe ninguna otra herramienta cuyo fin sea el mismo, no podemos hablar de que existan antecedentes del proyecto, aunque sí existe una herramienta de la que hablaremos más adelante que es perfecta para el objetivo que tenemos planteado. Se llama Graphiti [6].

1. 2. Objetivos del proyecto

El "diseño por contrato" propone la especificación precisa y verificable de los requisitos que un programa debe cumplir. Su propósito es obtener sistemas más fiables y robustos. Para ello, se definen contratos que hacen explícito el comportamiento esperado del software y a partir de los cuales es posible generar automáticamente aserciones o "funciones oráculo" para probar la corrección del sistema implementado.

En este Trabajo Fin de Grado se planteó desarrollar un entorno similar al existente (transML [4]), pero que permitiera la definición de contratos de manera gráfica, que en muchas ocasiones resulta más intuitiva que una representación textual. Para desarrollar el entorno se utilizaría Graphiti [6], que es un framework de Java para la construcción de herramientas gráficas.

1. 3. Estructura del documento

Pasamos a explicar la estructura que sigue este documento. El orden seguido es, al parecer del desarrollador, el orden lógico de lectura para que un usuario ajeno al desarrollo conozca bien qué se ha hecho, por qué se ha hecho y cómo funciona lo que se ha hecho.

- **Sección 2**: En esta sección hablaremos sobre las tecnologías que se han utilizado para llevar a cabo el propósito del proyecto.

- **Sección 3**: Aquí explicamos el modelo de dominio sobre el que está todo implementado. Es necesaria la comprensión de éste, puesto que está muy relacionado con el editor gráfico que se ha realizado.

- **Sección 4**: En el Análisis hablamos sobre la conexión entre los objetos del modelo de dominio y los objetos que dibujamos a través del editor gráfico. Mostramos su apariencia y comentamos cómo actuarán en su ejecución.

- **Sección 5**: En el Diseño detallamos qué es y cómo funciona Graphiti [6] y también lo hacemos sobre la estructura que hemos seguido a la hora de desarrollar la herramienta.

- **Sección 6**: En este apartado explicamos todo sobre la Implementación, qué clases tenemos, cuál es el fin de éstas y cómo están relacionadas entre ellas.

- **Sección 7**: Este apartado es un *tour* guiado sobre la ejecución y uso de nuestro proyecto. Aquí el lector verá cómo luce la herramienta en tiempo de ejecución.

- **Sección 8**: Esta sección presenta las conclusiones sobre el trabajo realizado y posibles líneas de trabajo futuro.

- **Sección 9**: Para terminar, aquí incluimos las referencias a los textos, páginas web y demás aspectos sobre los que se han basado algunas de las ideas expuestas en el documento.

2. Tecnologías a utilizar

Este apartado trata de las tecnologías utilizadas para llevar a cabo este proyecto. Así como también se hablará sobre el porqué de éstas.

2. 1. Estado del arte

El objetivo del proyecto es el de crear un editor gráfico que cumpla con una serie de funcionalidades relacionadas con los metamodelos. Hoy día, siempre existe un amplio abanico de posibilidades a la hora de elegir herramientas de desarrollo, y para el tipo de editor que se quiere construir, existen varias de ellas. Podemos destacar tres de ellas: Eugenia [7], GMF [8] y DSL Tools [9].

GMF [8] es un *framework* que contiene una serie de archivos de configuración (modelo de dominio, definición gráfica y definición de la herramienta) que juntos crean un plug-in para eclipse con una apariencia cuanto menos profesional. Ofrece las funcionalidades de imprimir, guardar imagen, *drag-and-drop*, ...

Eugenia [9] es un complemento de GMF [8]. Se encarga de generar los ficheros que hemos comentado anteriormente automáticamente a partir de un metamodelo de Ecore. Además también es posible continuar usando Eugenia para realizar el editor. Aparte de esa tarea que hemos explicado también nos da otras posibilidades.

Mientras que los dos anteriores estaban orientados a Eclipse, DSL Tools [9] funciona en Visual Studio. DSL Tools [9] nos permite la construcción personalizada de editores gráficos y la generación de código fuente usando anotaciones esquemáticas de dominio específico. Resumiendo, esta herramienta nos permite generar un editor gráfico de apariencia similar a un editor de UML.

Como hemos visto, existen varias herramientas de similar funcionalidad. En nuestro caso, el proyecto se realizará utilizando Graphiti [6] ya que en los requisitos del proyecto estaba estipulado así.

2. 2. Base de desarrollo

Para la realización de este proyecto hemos requerido de dos herramientas que han hecho posible cumplir con los requisitos de éste de la manera más sencilla y mejor valorada por el desarrollador. A continuación explicaremos más detalladamente esas herramientas:

- **EMF** (Eclipse Modeling Framework), tecnología de modelado [10].

Esta tecnología de modelado sustenta la base de Eclipse [5] cuando nos referimos a desarrollo dirigido por modelos, lo cual incumbe generación de metamodelos y autogeneración de código relacionado con el metamodelo creado. Además también nos ofrece la funcionalidad de interactuar con los modelos y metamodelos en tiempo de ejecución a través de sus librerías.

EMF [10] consta también de un entorno para salvar toda la información de los modelos en formato XMI (XML Metadata Interchange).

Un proceso a seguir sería crear un metamodelo usando las herramientas que EMF [10] dispone para tal cometido y autogenerar código a partir de este metamodelo creado para manejar las clases de éste en tu aplicación.

- **Eclipse** [5]

Eclipse es un IDE multiplataforma de código abierto cuya funcionalidad principal es la de entorno de desarrollo para diversos lenguajes. Como la gran mayoría de entornos de desarrollo, consta de resaltado de sintaxis y compilación de código (en tiempo real) así como la posibilidad de ejecución. La gran ventaja de Eclipse [5] es que además de su funcionalidad base, también ofrece la posibilidad de utilizar plug-ins (funcionalidades que se añaden a la base) lo cual hace que Eclipse tenga un campo de acción muy amplio. Algunos plug-ins que podemos poner de ejemplo son aquellos que sirven para llevar a cabo un control de versiones de las aplicaciones, entornos de prueba para realizar pruebas sobre las aplicaciones desarrolladas o el propio plug-in de la plataforma EMF [10] que hemos explicado.

- **Graphiti** [6]

Puesto que esta herramienta tiene una sección propia (sección 5.1) en la que se explica su funcionamiento y las posibilidades que ofrece, seremos breves. Graphiti [6] es un plug-in de Eclipse [5] que nos facilita la creación de editores gráficos de una manera rápida y sencilla.

3. Lenguaje de contratos para transformaciones

A continuación vamos a explicar el metamodelo base para la implementación del proyecto. La función básica del editor que hemos creado es dibujar ciertos objetos. En nuestro caso, esos objetos dibujados por el editor simbolizan a otros. Éstos otros objetos de los que estamos hablando son los que define el metamodelo ilustrado en la **Figura 1**.

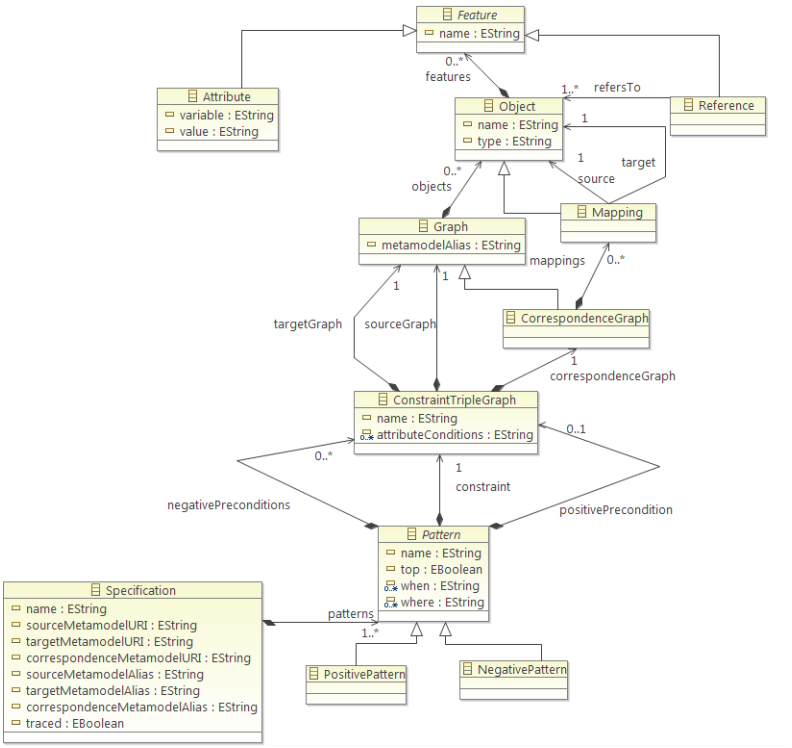


Figura 1. Analysis - metamodelo

El metamodelo es pequeño y consta de ciertos objetos que serán representados por los objetos que creamos en nuestra aplicación en determinados momentos (dependiendo del uso que hagamos de ella). Recordamos que el objetivo del proyecto es construir un editor que simbolice los diseños por contrato. Aunque va a ser explicado más en detalle a continuación, avanzamos sintetizadamente que sobre el objeto base llamado *Specification*, se añadirán *Patterns*, los cuales pueden contener *Objects* que pueden referenciar a otros *Objects* o contener una serie de *Attributes*. Así explicado, es difícil entender el modelo de la **Figura 1** por lo que ahora pasamos al detalle de la cuestión.

3. 1. Especificación

Esta clase es la raíz de todos los objetos que creamos (tanto gráficamente como en el modelo). Cada especificación representa la clase *Specification* del metamodelo explicando anteriormente y puede hacer referencia a tres metamodelos. Un primer metamodelo llamado *source* que simboliza el metamodelo de entrada y un segundo metamodelo llamado *target* que simboliza cómo debería quedar todo después de la transformación. Además, dependiendo del caso, podemos tener un tercer metamodelo llamado *correspondence* que simboliza un elemento auxiliar que sirve de apoyo para una mejor comprensión de la transformación (al poder unir objetos tanto del origen como del destino, aporta información sobre de qué objeto parte qué otro objeto) o también puede usarse para identificar las trazas de una transformación. Dependiendo de los metamodelos

especificados en cada especificación, el tipo de los objetos de los que hablaremos a continuación podrán contener unos valores u otros (podrán contener los tipos definidos en los correspondientes meta-modelos). Aún así, los metamodelos de los que hablamos también pueden ser nulos, en cuyo caso el tipo definido para los objetos podría ser cualquier (es decir, no se realiza comprobación de los tipos). Según vayamos explicando las demás partes del modelo, los conceptos irán conectándose y será más sencillo comprenderlo. Los atributos de los que consta cada especificación son:

- ✓ **name:** Nombre de la especificación.
- ✓ **sourceMetamodelURI:** URI del metamodelo de origen.
- ✓ **targetMetamodelURI:** URI del metamodelo de destino.
- ✓ **correspondenceMetamodelURI:** URI del metamodelo de correspondencia.
- ✓ **sourceMetamodelAlias:** Alias o nombre del metamodelo de origen.
- ✓ **targetMetamodelAlias:** Alias o nombre del metamodelo de destino.
- ✓ **correspondenceMetamodelAlias:** Alias o nombre del metamodelo de correspondencia.
- ✓ **traced:** Si este atributo vale true, quiere decir que los patrones de la especificación incluirán (tanto gráficamente como internamente) un modelo de correspondencia relacionando el modelo de entrada con el modelo de salida.

3. 2. Patrones: Patrones Positivos y Patrones Negativos

Cada especificación está compuesta de patrones (representados por la clase *Pattern* del metamodelo). Un patrón indica cómo debería ser la salida de la transformación partiendo de una configuración determinada de la entrada. Cada patrón tiene asociado un grafo triple, que a su vez está formado por dos grafos (representando un grafo de entrada y otro de salida) y un grafo de correspondencia (para el modelo de correspondencia). Cuando el atributo traced de la especificación es falso, el grafo de correspondencia está vacío. Cada Patrón simboliza una propiedad o cláusula del contrato. Además, éstos pueden tener una precondición positiva y varias precondiciones negativas. Estas precondiciones son condiciones adicionales para el patrón. El atributo del que consta cada patrón es:

- ✓ **name:** Nombre del patrón.

De la clase *Pattern* heredan otras dos llamadas *PositivePattern* y *NegativePattern*. Es un mero hecho de distinción ya que estas dos clases son sintácticamente iguales que la clase *Pattern*, salvo que esta última es abstracta y son esas dos las que de verdad se usan a la hora de utilizar el metamodelo. Aún así significan cosas diferentes. Los patrones positivos expresan que si sucede algo en la entrada, debe ocurrir algo en la salida mientras que los patrones negativos expresan que si sucede algo en la entrada, no debe suceder algo en la salida.

Los patrones solo pueden ser creados en el propio diagrama. Si se intenta crear un patrón dentro de otro o de cualquier otra figura, el editor no nos dejará hacerlo.

3. 3. Grafos Triples

Los grafos triples representan a la clase ConstraintTripleGraph del metamodelo. Desde el punto de vista gráfico también podrían ser llamados precondiciones aunque en este apartado nos dedicamos a explicar el metamodelo. Como la palabra "Triple" que podemos encontrar dice, esta clase es en realidad una trinidad de grafos. Cuando explicamos lo que era una especificación hablamos de tres metamodelos. El número tres se repite y no es casualidad. Cada grafo del grafo triple está asociado con los metamodelos de las especificaciones. Como podemos ver en la **Figura 1**, esta clase consta de 3 grafos cuyos nombres son *sourceGraph*, *targetGraph* y *correspondenceGraph* (éste último solo tiene sentido cuando el valor del atributo traced de *Specification* es verdadero). Todavía no sabemos bien que es un grafo pero sabemos que un grafo triple contiene tres de ellos. El primero está relacionado con el metamodelo de entrada, el segundo con el metamodelo de salida, el tercero aporta información adicional a la transformación (si existiera este tercero) y todos ellos establecen las normas de la transformación del patrón. Los atributos de los que consta cada *ConstraintTripleGraph* son:

- ✓ **name:** Nombre del grafo triple.
- ✓ **attributeConditions:** Aporta unas normas en forma de texto que se complementan con las normas de cada grafo.

3. 4. Grafos y Grafos de Correspondencia

Un grafo es un objeto de información que contiene unos elementos llamados objetos (mediante estos elementos, es como el grafo aporta información) que nos informan del estado de una determinada parte de un determinado modelo. Es la representación de la clase *Graph* del metamodelo. Dicho así no parece aportar mucho pero en el contexto de este proyecto cumple una función elemental. Aunque podemos tener tres grafos dentro de un grafo triple, nos fijaremos sólo en el grafo de origen y en el de destino (el grafo de correspondencia simplemente sirve de apoyo para comprender de dónde viene un objeto del destino). El primer grafo nombrado es el que contiene la información del modelo de entrada y el grafo de destino contiene la información relevante con la configuración de objetos esperada al hacer la transformación. Como bien hemos ido viendo en las secciones anteriores, para entender una clase es necesario entender las clases que esa clase utiliza. Cuando los objetos sean analizados en detalle, el metamodelo podrá ser entendido casi en su completitud. El atributo del que consta cada *Graph* es:

- ✓ **metamodelAlias:** Nombre del metamodelo asociado al grafo.

Aparte de la clase *Graph*, podemos encontrar otra clase que hereda de esta misma. Hablamos de la clase *CorrespondenceGraph* (o grafo de correspondencia). Repitiendo lo ya dicho, además de los metamodelos de origen y de destino, existe otro metamodelo de correspondencia. También hemos dicho que los grafos son elementos de información del metamodelo. Un grafo de correspondencia no es más que un grafo que hace referencia al metamodelo de correspondencia. Por esto, y como explicaremos más adelante, este objeto puede contener mappings (objetos que un simple grafo no contiene).

3. 5. Objetos y Mappings

Cada objeto es la unidad de información contenida en los grafos. Representan a la clase *Object* del metamodelo. Cada objeto simboliza un elemento del metamodelo correspondiente con el grafo en el que éste se encuentre. Además, pueden referenciar a otros objetos dentro del mismo grafo y proporcionar información adicional a través de sus propios atributos. En el grafo de origen tendremos objetos que representan una serie de clases del metamodelo correspondiente mientras que en el grafo de destino aparecería la salida esperada representada con objetos basados en elementos del metamodelo cargado en ese otro grafo. Los atributos de los que consta cada *Object* son:

- ✓ **name:** Nombre del objeto.
- ✓ **type:** Representa el tipo del objeto en base al metamodelo correspondiente del grafo. Si no hay ningún metamodelo cargado, el tipo puede ser cualquiera que el usuario desee (ya que es un *String* y no tiene ninguna limitación). En caso contrario y afirmativo, el tipo del objeto debería ser el mismo que el tipo de alguno de los elementos del metamodelo con el cual se quiere asociar ese objeto.

En el caso de que el atributo traced de *Specification* valga true, como ya hemos dicho, contaríamos también con el grafo de correspondencia. Éste, y sólo éste, es capaz de contener un mapping (son iguales que los objetos salvo que la clase que los mappings representan, llamada *Mapping*, hereda de *Object*). Un *mapping* es un objeto con la única diferencia de que éste puede referenciar objetos de diferentes grafos (aunque sólo de un mismo grafo triple). Puede contener hasta tres referencias: un objeto de su mismo grafo y otros dos de los grafos origen y destino.

Los objetos solo pueden ser creados dentro delos patrones o de las precondiciones. Si intentamos crear un objeto en cualquier otro lugar el editor nos lo impedirá. Con los mappings sucede lo mismo salvo que dentro de esos patrones y esas precondiciones, el espacio de creación se les limita a la columna central (grafo de correspondencia) si la hubiera.

3. 6. Atributos y Referencias

Tanto los atributos y referencias representan la clase *Feature* la cual es una clase abstracta de la que heredan *Attribute* y *Reference*, que aporta características a los objetos. El atributo del que consta cada *Feature* es:

- ✓ **name:** Nombre de la *Feature*.

Los atributos representan los atributos de cada objeto, valga la redundancia. Un atributo es una característica que simboliza un campo del elemento del metamodelo representado por el objeto. En el caso de que haya un metamodelo cargado, el nombre del atributo deberá coincidir con el nombre del elemento del objeto del metamodelo. En caso contrario, el atributo puede tener cualquier nombre. Los atributos de los que consta cada *Attribute* son:

- ✓ **value:** Valor del *atributo*.

- ✓ **variable:** También representa el valor del atributo pero haciendo referencia a una variable. Sólo uno de los dos atributos será utilizado.

Por último, nos queda hablar de las referencias. Como ya dijimos antes, un objeto puede referenciar a otro dentro del mismo grafo. Esa referencia se lleva a cabo mediante el uso de esta clase. El objeto contendrá una instancia de una referencia la cual referenciará a otro objeto.

4. Análisis

En este apartado vamos a explicar los elementos gráficos que el usuario podrá crear y manipular con el editor.

La distribución gráfica del editor es sencilla y consta de dos partes. Por un lado tenemos un lienzo que llamaremos diagrama en el cual se crearán todos los demás objetos y por otro lado tenemos la paleta. La paleta es una región situada a la derecha del diagrama, la cual nos lista todos los elementos que pueden ser añadidos a éste (o a algún otro elemento dentro del diagrama).
A continuación pasamos a explicar en detalle cada uno de los elementos que pueden ser dibujados por el usuario. Comentaremos la relación de éstos con el metamodelo que analizamos en la sección anterior y también explicaremos cómo se comportan esos objetos cuando se mueven o modifican, las hojas de propiedades asociadas a ellos e ilustraremos la forma de éstos.

4. 1. Diagrama

El diagrama se crea directamente al ser un proyecto de Graphiti, eligiendo como tipo de diagrama el que hemos creado nosotros, que en este caso se llama *ContractSpecification*. Para conseguirlo es preciso tener instalado en Eclipse el plug-in desarrollado.

Diagramas solo hay uno y el usuario no puede crear más o modificar su forma. Está ahí desde el principio y en él se irán añadiendo otras figuras. Este elemento está asociado con la clase *Specification* ya analizada en la sección 3.1. Como bien veremos más adelante, al igual que una especificación contenía patrones dentro, el diagrama contendrá las figuras que representan a estos patrones. Se establece un paralelismo entre los objetos físicos que dibujamos y los objetos instanciados del metamodelo. Cuando mediante el uso de la paleta dibujamos un patrón en el diagrama suceden dos cosas: aparece una figura que representa un patrón (ya veremos cómo son a continuación) en el diagrama y un objeto Pattern se agrega a la lista de patrones de la clase *Specification*.

Al crear el diagrama se debe preguntar al usuario si quiere una especificación con trazas o no, lo que fija el tipo de patrones que contendrá la especificación (con trazas o sin trazas). Eso no puede cambiarse posteriormente.

- **Hoja de propiedades:** Cuando hacemos *click* sobre el diagrama aparecerá la vista de propiedades de Eclipse (si la tenemos visible) en la cual podemos ver y modificar algunos datos de la Specification. Estos datos a los que nos referimos son los tres metamodelos de origen, destino y correspondencia. Por ejemplo, podemos cambiar el metamodelo cargado en origen o simplemente hacer que el origen no tenga ningún metamodelo (es igual para los otros dos casos). Además, aparece cargada la lista de metamodelos registrados en el Sistema, para facilitar su selección.

4. 2. Patrones: Patrones Positivos y Patrones Negativos

Como ya hemos dicho, cuando creamos un patrón suceden dos cosas: se crea la figura y se crea el objeto de dominio por debajo. Como en el caso de la especificación, la figura del patrón está ligada con el objeto de dominio y está implementado todo para que cuando modificamos un elemento de la figura, se modifique ese campo en el objeto de dominio también. Esto es así para todos los componentes que dibujamos en el diagrama o dentro de elementos del diagrama por lo que no volverá a ser explicado en secciones siguientes. Puesto que los patrones positivos y negativos son exactamente iguales, explicaremos los patrones en general. La diferencia es que los positivos son de color **verde** y los negativos de color **rojo**. A cada patrón es posible añadirles una precondición positiva y varias precondiciones negativas (cuya apariencia es la misma que la de sus respectivos patrones). Además, los patrones pueden ser colapsados y extendidos por el usuario, ya que al tratarse de elementos grandes, pueden facilitar la visualización del diagrama en algún momento .

- **Layout:** Dentro de un patrón se pueden añadir objetos y mappings (sólo en el caso de que el atributo traced de la clase *Specification* sea true). La figura de un patrón puede ser redimensionada (agrandada o empequeñecida) y tiene unos tamaños mínimos. Como veremos ahora, contiene unas líneas horizontales y verticales que se adaptan al tamaño del patrón cuando lo modificamos. La línea vertical puede ser movida lateralmente sin salir de la figura.

- **Hoja de propiedades:** Aquí podemos observar y modificar el nombre del patrón y las condiciones de los atributos de éste. Más adelante veremos la representación gráfica de los grafos triples que es igual que ésta. Aquí debemos decir que un patrón no debe tener el mismo nombre que otro. El editor nos lo notificará con un icono de error.

➢ **Pictograma:**

4. 3. Patrones con Traza: Patrones con traza Positivos y Patrones con Traza Negativos

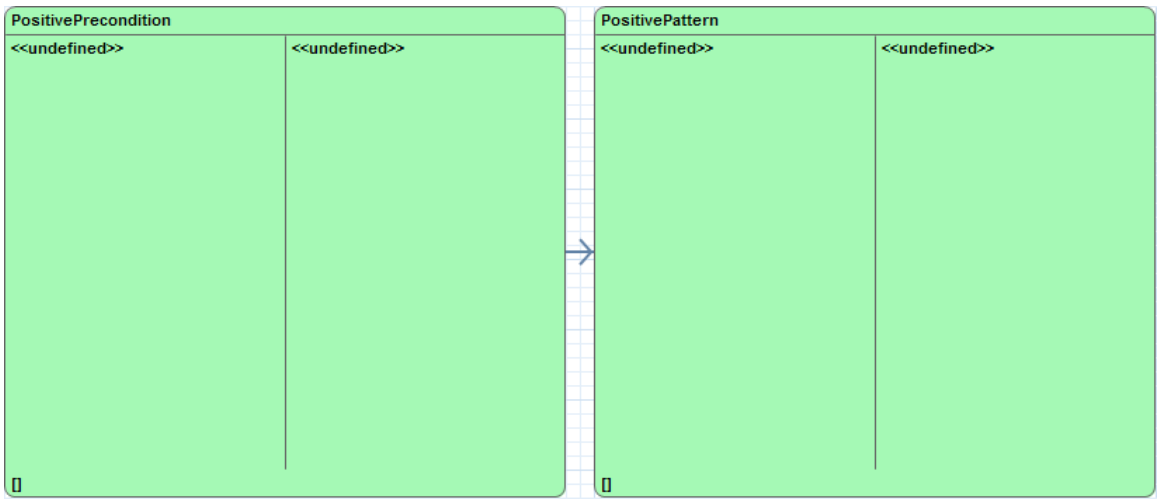
Cuando al crear una especificación se indica que es trazada (atributo traced), los patrones mostrarán un grafo intermedio (grafo de correspondencia). Los patrones trazados se comportan igual que los patrones pero tienen una columna extra en el medio. Ambos gráficamente se representan distinto que los patrones sin trazas, internamente el objeto de dominio que se usa para almacenarlos es el mismo (*Pattern*).

➢ **Pictograma:**

4. 4. Precondiciones Positivas y Precondiciones Negativas

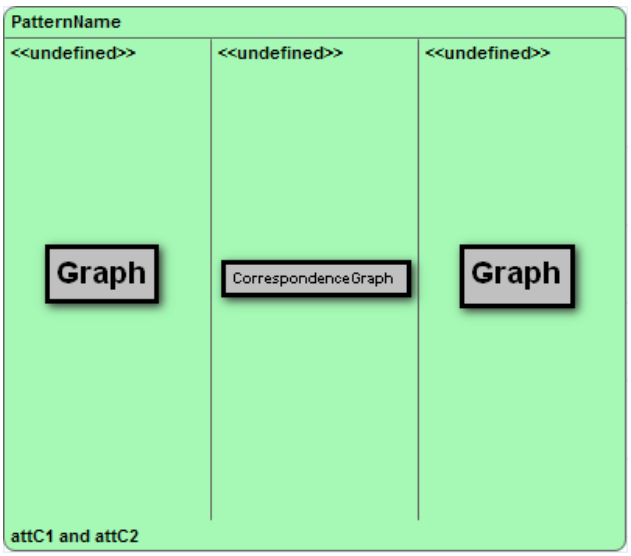
La representación de una precondición y de un patrón es exactamente la misma. La única diferencia es la siguiente: cuando creamos un patrón desde la paleta se crea en el diagrama. Si añadimos una precondición al patrón, lo que sucede es que aparece otra caja igual que la del patrón al lado de éste primero unidas por una flecha. Visualmente son dos patrones aunque en realidad el patrón es el que hemos creado con la paleta mientras que el de al lado es una precondición de éste.

- **Pictograma:**



4. 5. Grafos y Grafos Triples

Estos elementos no tienen representación gráfica individual. Aún así, a continuación señalaremos qué es un Graph dentro de un *Pattern* o *ConstraintTripleGraph* (también nos referimos al caso *traced*):



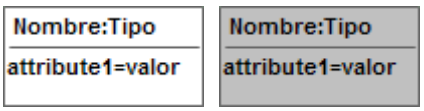
4. 6. Objetos y Mappings

Antes de nada decir que un mapping se comporta exactamente igual que un objeto visualmente salvo porque el color es **diferente** (para distinguirlos). Los mappings tienen una funcionalidad aumentada respecto a los objetos. Éstos pueden relacionarse con objetos de otros grafos (dentro del mismo grafo triple) mientras que los objetos solo pueden hacerlo con objetos del mismo. Los objetos sólo pueden ser creados dentro de los patrones mientras que los mappings sólo en la columna del medio si la hubiera (ambos se crean utilizando la paleta). Repitiendo el concepto del que hablamos anteriormente, recordamos que al dibujar un objeto también se está añadiendo un elemento de tipo *Object* al *Pattern* del modelo de dominio.

- **Layout:** El objeto puede redimensionarse con el ratón y la línea horizontal de la que consta éste lo hace también.

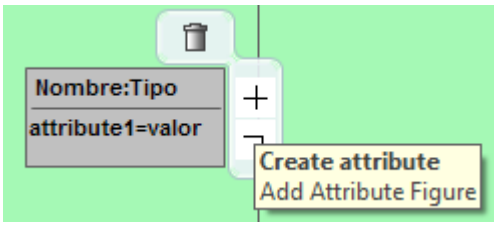
- **Hoja de propiedades:** Aquí podremos ver y modificar el nombre del objeto y el tipo. Este último campo podrá ser modificado mediante texto si no hay ningún metamodelo cargado en la especificación para el grafo en que se encuentra el objeto o bien mediante un desplegable si lo estuviere. No pueden existir objetos con el mismo nombre dentro de un mismo patrón y tanto el nombre como el tipo deben tener algún valor. En caso contrario el editor nos lo notificaría con un icono de error.

- **Pictograma:**



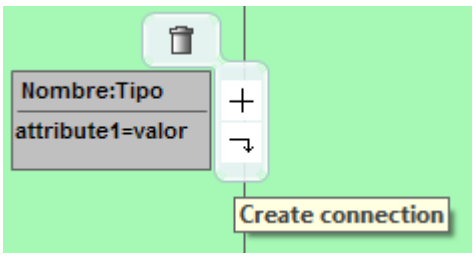
4. 7. Atributos

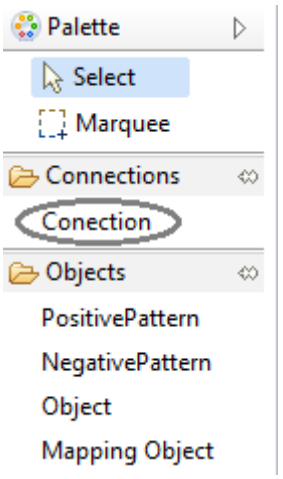
El atributo se añade al objeto (tanto a la figura como al objeto de dominio) clicando el botón de "Create attribute" en el menú contextual del objeto. Este atributo debe cumplir unas normas. Si el objeto en el que se encuentra no es de un tipo determinado que se encuentre en el metamodelo cargado (debe haber uno cargado en ese caso) puede tener cualquier tipo. En caso de que haya un metamodelo cargado y el objeto en el que se encuentra sea de un tipo determinado del metamodelo, este tipo debe coincidir con alguno de los tipos de los atributos que puede tener ese objeto. En caso contrario el editor nos avisaría con un icono de error.



4. 8. Referencias

La referencia se añade al objeto (tanto a la figura como al objeto de dominio) clicando el botón de "Create connection" en el menú contextual del objeto o bien usando la conexión de la paleta. Si el objeto del que parte la referencia no es de un tipo determinado que se encuentre en el metamodelo cargado (debe haber uno cargado en ese caso) puede tener cualquier nombre. En caso de que haya un metamodelo cargado y el objeto en el que se encuentra sea de un tipo determinado del metamodelo, el nombre de la referencia debe coincidir con alguna de las referencias que pueden salir de ese objeto. En caso contrario el editor nos avisaría con un icono de error.





5. Diseño

En esta sección analizaremos el framework principal que hemos utilizado para desarrollar el proyecto, ya que entendiendo bien como éste funciona, facilita el entendimiento de cómo hemos implementado la aplicación. Además también hablaremos sobre cómo hemos estructurado todo y sobre algunas decisiones de diseño tomadas.

5.1. Graphiti

El objetivo de este proyecto es el de realizar un editor gráfico para definir de manera gráfica contratos para un tipo de software concreto: los programas de transformación de modelos.

Cuando se trata de construir un editor gráfico que involucra la gestión y control de un modelo de dominio por debajo, surgen dos conceptos: EMF [10] encargado de gestionar el modelo y GEF [11] encargado de construir el editor. En realidad EMF [10] es un framework bastante avanzado y con mucha experiencia que facilita mucho la gestión de modelo. El problema está en GEF [11]. Es un framework a un no muy alto nivel que requiere de una gran experiencia y conocimiento sobre él para poder hacer un buen uso. Aquí es donde surge la herramienta Graphiti [6], la cual facilita con creces el manejo de figuras y gráficos para poder construir un editor con mucha más facilidad.

Por lo que acabamos de explicar, elegimos este framework para el desarrollo de la aplicación. A continuación observaremos en profundidad y detalle cómo está organizado Graphiti [6] y cómo trabaja.

Dividiremos el análisis del framework en dos partes. Por un lado explicaremos cómo gestiona los pictogramas (figuras, líneas, ...) que componen el editor en relación con el modelo y por otro veremos cómo maneja estos pictogramas en relación con el uso que se le da al editor.

Podéis encontrar un tutorial muy completo que explica más en detalle el funcionamiento de Graphiti [6] y aporta el conocimiento necesario para empezar a desarrollar usando Graphiti en la página web del framework [12].

5.1.1. Representaciones gráficas

Como podemos ver a continuación en la **Figura 2** existe un *link* (conexión) entre todos los elementos que dibujamos en el diagrama y los objetos de dominio. El objeto de dominio puede ser enlazado a varios elementos (existe un método llamado “link()” con el cual el desarrollador enlaza los elementos según necesite) y en nuestro caso, cada objeto de dominio esta enlazado con el contenedor (*ContainerShape*) de la figura. De esta forma si clicamos sobre un patrón en el diagrama (recordamos que el contenedor de la figura, que es lo que obtienes al clicar en ella, esta enlazado con el patrón) podemos obtener el objeto de dominio y actuar sobre él gracias a Graphiti [6].

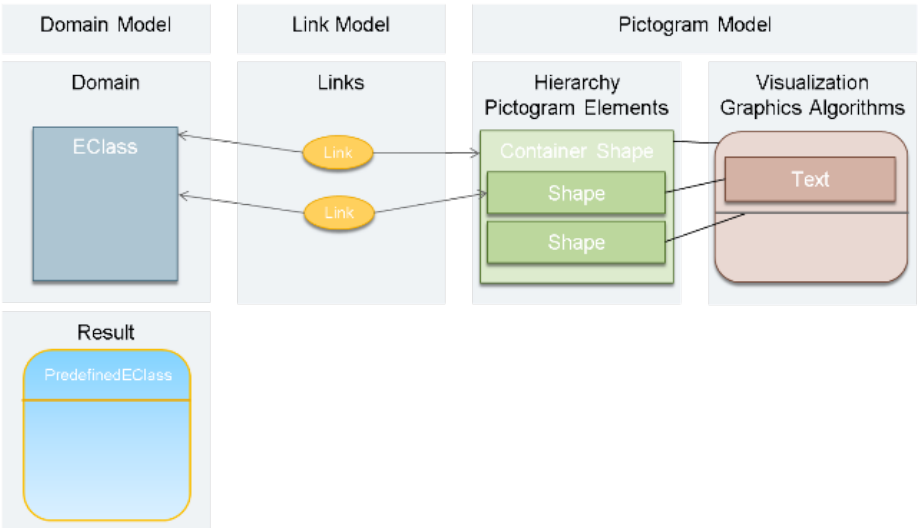


Figura 2. Link de los elementos utilizados. Extraída de [8]

5. 1. 2. Características del editor

Graphiti [6], como ya hemos dicho, nos facilita la elaboración de un editor gráfico a través de GEF [11]. Por defecto existen una serie de funcionalidades que actúan sobre los elementos del editor que necesitan ser ampliadas o adaptadas para cada elemento sobre el cual queremos que actúen. Dicho de otro modo, existen unas clases abstractas sobre las cuales debemos extender nuestras propias clases adaptadas a cada elemento y adaptar la funcionalidad. Algunas funcionalidades como por ejemplo borrar o modificar el tamaño de los elementos, están implementados por defecto. Esto quiere decir que si yo dibujo un cuadrado en el diagrama puedo eliminarlo y modificar su tamaño sin crear ninguna clase extra. Ésto sería sólo necesario si por ejemplo queremos que al agrandar el cuadrado se agrandara también una línea que éste tenga en el interior. Otros casos, como añadir o crear, aunque sí es verdad que ya está implementado el caso por defecto, necesitamos sí o sí crear estas clases y extenderlas para que la funcionalidad pueda llevarse a cabo.

A continuación podemos observar en la **Figura 3** que existe una clase llamada *FeatureProvider* que es la que gestiona todas las *features* (funcionalidades) ya sean añadir, crear, eliminar, actualizar, ...

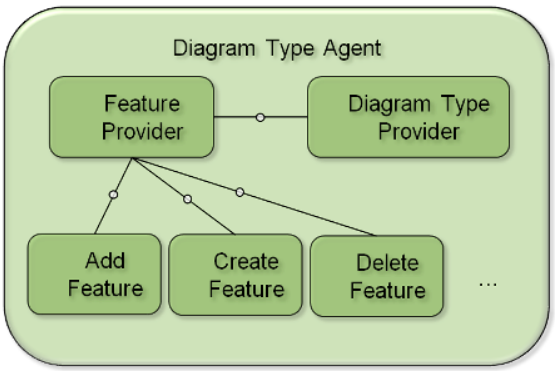


Figura 3. Estructura de las funcionalidades. Extraída de [8]

A continuación pasamos a analizar cada funcionalidad por separado para entender mejor cómo funciona este framework:

- **Add:** Extiende de *AbstractAddShapeFeature*. La funcionalidad de esta *feature* es la de dar forma al objeto. En ésta se define qué forma tendrá, qué línea o polígono estará dentro de qué otro y además se definirá si se desea, con qué objeto de dominio estará enlazada.

  Ya explicamos en la **Figura 2** cómo funciona o cómo utilizamos las conexiones entre el pictograma y el objeto de dominio. Aún así, ya que en esta *feature* es donde se produce ese enlace, adjuntamos otra imagen que representa la misma acción de otra manera para que quede más claro.

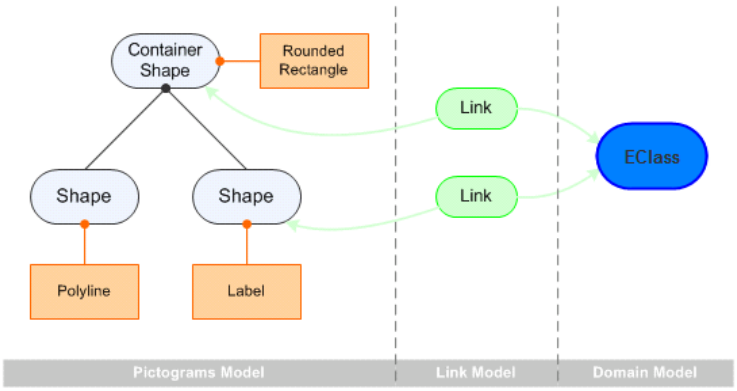


Figura 4. Estructura de la conexión. Extraída de [8]

  En nuestro caso, la representación de un *Pattern* muestra entre otras cosas el nombre de éste. Al tener enlazado el pictograma con el objeto de dominio podemos modificar el nombre tanto de la imagen como del último citado.

- **Create:** Extiende de *AbstractCreateFeature*. En esta *feature* es donde creamos el objeto de dominio y lo añadimos dentro del modelo de dominio donde proceda. Además el usuario puede restringir en esta clase dónde puede ser creado el elemento y dónde no.

- **Delete:** Extiende de *DefaultDeleteFeature*. Esta funcionalidad viene implementada por defecto. Puedes elegir un elemento y borrarlo. Aún así, es posible que el usuario desee que aparte de realizar un borrado básico, realizar otras acciones (por ejemplo un borrado en cascada de "X" elementos). Para esto existe esta *feature*.

- **Update:** Extiende de *AbstractUpdateFeature*. Esta *feature* primero mira si es necesario refrescar algo del elemento. En caso afirmativo realiza los cambios sobre éste que el desarrollador haya creído oportunos y haya implementado en la *feature Update* de ese elemento.

- **Move:** Extiende de *DefaultMoveShapeFeature*. La funcionalidad de esta feature es sencilla. Deja o impide moverse al elemento que quiere ser desplazado. Esto hace, por ejemplo, que no deje sacar una figura contenida dentro de otra o que sólo pueda moverse horizontalmente.

- **Layout:** Extiende de *AbstractLayoutFeature*. Esta *feature* sirve para actualizar la forma o formas de "X" elementos contenidos dentro de otro (y de éste mismo) que ha sido modificado de alguna manera. Volvemos a un ejemplo que pusimos antes. Si agrandamos un rectángulo, esta *feature* lo que hará es que verá que el elemento ha sido modificado y se ejecutarán las acciones que el desarrollador haya creído conveniente y haya incluido en la *feature Update*. Posiblemente, agrandar en igual medida una línea horizontal contenida en el elemento modificado entre otras.

- **Properties:** Las figuras pueden tener asociado a ellas una hoja de propiedades que muestran y dejan modificar información relevante con ellas. Si un elemento quiere tener esa hoja de propiedades lo que tiene que pasar es que exista una clase *Filter* para ese elemento que extienda de *AbstractPropertySectionFilter* (nos dice si la figura que estamos analizando es sobre la que hay que mostrar la hoja de propiedades) y una clase *Section* que extienda de *GFPropertySection* donde se crea la hoja de propiedades (campos de los que constará, etiquetas, ...).

5.2. Estructura del proyecto

En este apartado comentaremos cómo hemos organizado nuestro proyecto. La estructura es bastante elemental ya que Graphiti [6] internamente de por sí lo es. Hablaremos de los paquetes que hemos usado y cómo hemos dividido las clases en ellos.

5.2.1. Estructura de paquetes

Durante el desarrollo del proyecto se ha optado por juntar las clases con funcionalidades similares o relacionadas con un mismo aspecto en un mismo paquete. El mismo criterio se ha seguido con los paquetes que se encuentran dentro de otros paquetes. Primero explicaremos la estructura del proyecto a alto nivel y luego entraremos en detalle para analizar el contenido de cada paquete si fuera necesario.

La jerarquía de paquetes es llana ya que las funcionalidades dentro de Graphiti [6] están muy diferenciadas y por lo tanto existe un paquete para cada tipo. Aún así, en nuestro caso tenemos paquetes situados dentro de otros paquetes. En el primer nivel de paquetes encontramos, como bien podemos ver en la **Figura 5**, la raíz de los paquetes.

Figura 5. Paquete raíz

En el paquete *contractSpecification* simplemente encontramos la clase activadora del plug-in. Ahora es cuando veremos dónde se encuentra la verdadera funcionalidad del proyecto. Dentro de la raíz podemos encontrar los siguientes paquetes:

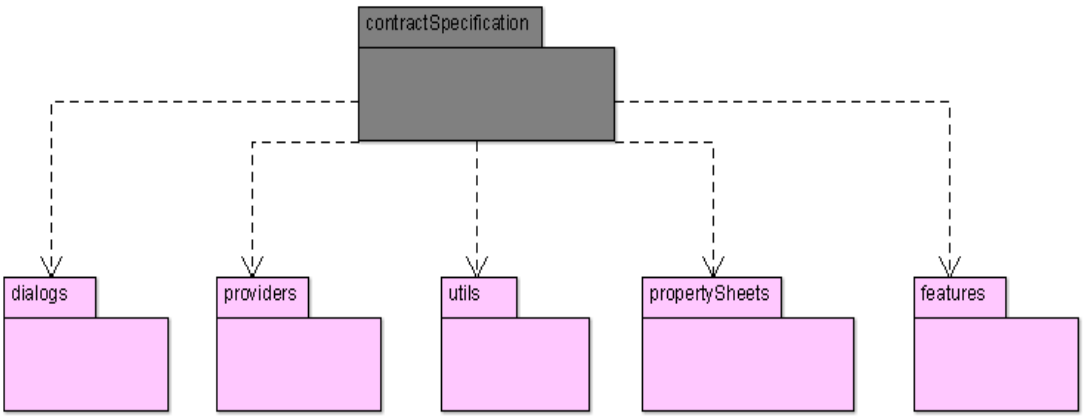


Figura 6. Segundo nivel de paquetes del proyecto

- **dialogs:** En este paquete se encuentran las clases relevantes con los *dialogs* o paneles que se usarán en el programa.

- **providers:** Aquí se definen los cuatro *providers* de que dispone nuestro proyecto los cuales analizaremos más adelante. Estos son el *DiagramTypeProvider* (*provider* principal que se encarga del arranque del proyecto), *FeatureProvider* (encargado de la gestión de funcionalidades en la aplicación), *ImageProvider* (gestiona una serie de iconos definidos por el desarrollador) y *ToolBehaviourProvider* (encargado de proveer las diferentes acciones que se pueden encontrar en los menús contextuales y también de los iconos de error o de *warning*).

- **utils:** Este paquete auxiliar consta de clases que facilitan el desarrollo del proyecto. Ya sean una clase donde se establecen los estilos personalizados de cada figura, otra con métodos generalizados de gestión de modelos, ... Estas clases son todas clases de "ayuda" y se reúnen en este paquete adicional.

- **properySheets:** Todas las hojas de propiedades que utilizamos al clicar sobre una figura o sobre el propio diagrama están definidas aquí.

- **features:** Este paquete contiene las clases que aportan las diferentes funcionalidades de que consta nuestra aplicación. Serán explicadas con más detalle a continuación.

El siguiente nivel de paquetes se encuentra dentro del paquete *features*. Se ha creado un paquete para cada tipo de acción que se puede realizar en el diagrama. La **Figura 7** muestra cada uno de esos paquetes, que se explican a continuación.

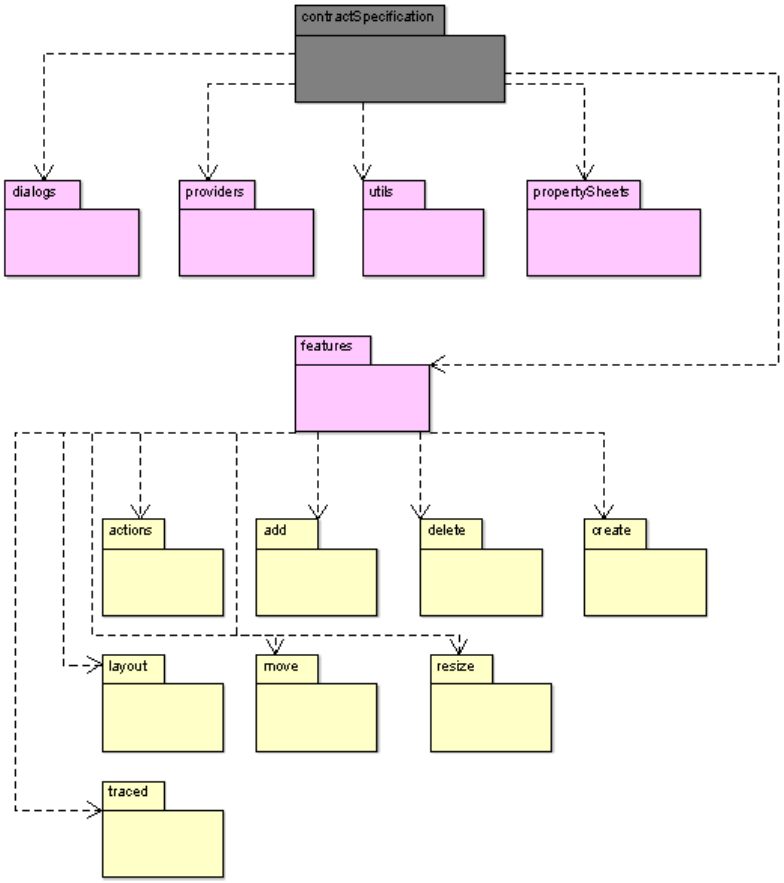


Figura 7. Tercer nivel de paquetes del proyecto

Ahora, aunque cuando hablemos de la implementación repetiremos conceptos sobre las clases incluidas en cada paquete, daremos una pequeña y resumida noción de qué tipo de clases se encuentran en éstos. Además esto está muy conectado con los conceptos explicados en la sección 5.1.2.

- **actions:** Aquí están las clases cuya funcionalidad se ejecuta cuando pulsamos uno de los botones contextuales de cada figura (por ejemplo el de colapsar o el de añadir un atributo).

- **add:** Aquí se definen las representaciones gráficas de cada figura que puede ser dibujada en el diagrama.

- **delete:** Dispone de las clases que especifican qué pasa cuando borramos una figura determinada.

- **create:** Las clases que contiene este paquete son las que crean el objeto de dominio.

- **layout:** Aquí están las clases que modifican la apariencia e información de unos determinados elementos en cierto momento.
- **move:** Puesto que todas las figuras se pueden mover, aquí se especifica la acción para otras determinadas que no lo hacen de la manera habitual.

- **resize:** Puesto que todas las figuras se pueden redimensionar, aquí se especifica la acción para otras determinadas que no lo hacen de la manera habitual.

- **traced:** Ya comentamos que hay un caso en el que los patrones contienen una columna extra (cuando el atributo traced de *Specification* vale *true*). En este caso, tanto la apariencia de algunas figuras, como la forma en que se mueven, la forma en que se actualizan, ... son diferentes. Por ello en este subpaquete se detallan otros subpaquetes que en algunos casos sus clases mantienen herencia con las del paquete correspondiente de su padre (ya que las funcionalidades son las mismas). En otros casos se han creado las clases independientes debido a que facilitaba las cosas (se hablará de ello en la siguiente sección). También algunas clases del paquete *features* se utilizan en el caso *traced* por lo que no se crea de nuevo esa clase en el paquete del que estamos hablando.

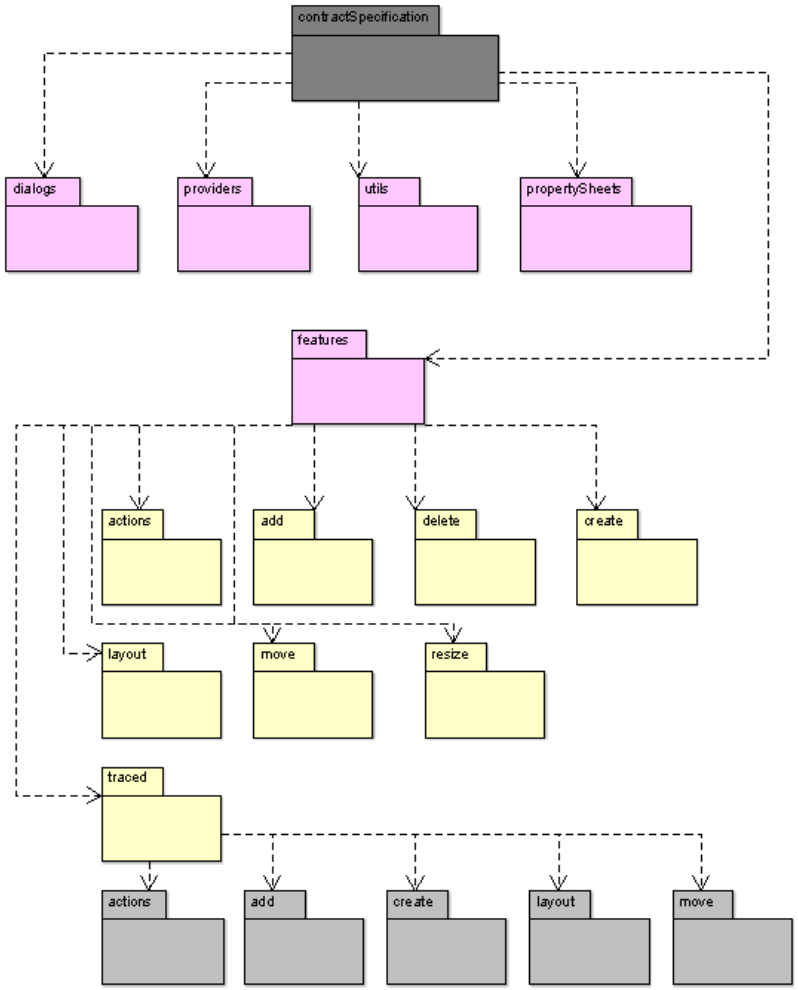


Figura 8. Cuarto nivel de paquetes del proyecto

6. Implementación

Al explicar la estructura del proyecto, uno puede hacerse a la idea de cómo ha sido implementado y qué tipo de clases se encuentran dentro de los paquetes. En esta sección, a diferencia de la anterior, no nos centraremos en los paquetes sino en lo que hay dentro de ellos.

Primero comentaremos cómo hemos configurado el entorno de desarrollo para trabajar y más adelante entraremos en detalle con el contenido de cada paquete explicado.

6. 1. Entorno de desarrollo

Como ya dijimos con anterioridad la plataforma sobre la que hemos desarrollado nuestro proyecto es Eclipse. En ella creamos un nuevo proyecto de tipo plug-in (ya que el objetivo era extender la funcionalidad base de que parte la plataforma). Aparte de la

funcionalidad base de Eclipse [5] se han requerido de otros servicios que no nos proporciona esta plataforma de serie. Ha sido necesaria la instalación de otros plug-ins como bien han sido:

- **Graphiti** 0.10.0.v20130612-0936
- **EMF** 2.9.1.v20130902-0605

Podemos observar estos plug-ins de los que hablamos si miramos las dependencias que necesita el proyecto de Eclipse así como los paquetes importados:

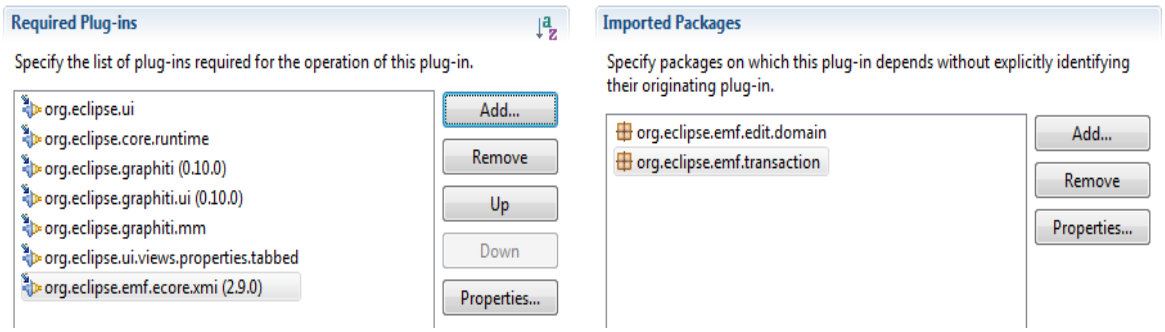


Figura 9. Dependencias y paquetes importados.

6. 2. Features del editor

Todas las features de las que hablamos antes tienen su implementación. Aquí hablaremos de ello y podréis relacionar qué *feature* está relacionada con qué elemento del diagrama y del modelo de dominio.

6. 2. 1. Create

Las clases *create* las cuales extienden de *AbstractCreateFeature* son las que hacen que el objeto aparezca en la paleta y comienzan la creación de éste en el diagrama (así como en el modelo de dominio). Contienen un método que, dependiendo de dónde el usuario quiera crear el objeto, nos lo permite o no. Recordamos que podemos crear tres elementos: patrones (positivos y negativos), objetos y conexiones. Los patrones sólo pueden crearse en el diagrama (si intentamos crear un patrón dentro de otro patrón el editor no nos dejará), los objetos solo pueden crearse dentro de los patrones (no nos dejará crearlos en otro sitio como ya hemos comentado) y las conexiones (en el caso básico que no contiene el grafo de correspondencia) sólo pueden realizarse entre objetos dentro del mismo grafo.

En el caso *traced*, también disponemos del create de los mappings, los cuales sólo pueden ser creados dentro del grafo de correspondencia. Además las conexiones entre mappings y objetos también están limitadas. Sólo nos dejará conectar con el destino y el origen si esa conexión no existe.

La creación de los patrones con el grafo de correspondencia es una extensión de la creación de los patrones normales.

Puesto que el patrón donde incluimos los objetos es un simple rectángulo con una línea vertical (no dispone de dos lugares donde poder instanciar el objeto), esta clase se encarga también de reconocer la posición en la que se ha creado el objeto en relación a la línea vertical para saber si se ha creado en el origen o en el destino (o en el grafo de correspondencia).

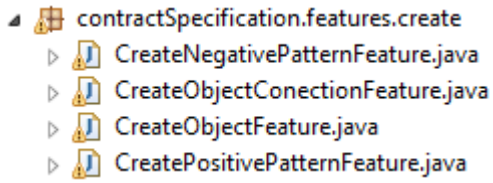


Figura 10. Clases Create del caso básico.

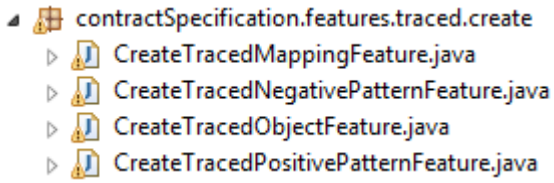


Figura 11. Clases Create del caso traced.

6. 2. 2. Add

Recordamos que esta *feature* es la encargada de dar forma a los objetos que aparecen en la paleta. Cada uno de ellos tiene su propia representación gráfica implementada en su clase *Add*. Además de tener relación con las figuras de la paleta, la clase *Add* también lo hace con las conexiones. Cuando hablamos de una figura, se detalla la apariencia de ésta y cuando nos referimos a una conexión se detallan los decoradores que irán unidos a ella.
Ahora podemos observar qué clases han sido implementadas y sobre ello comentaremos algunos aspectos relevantes.

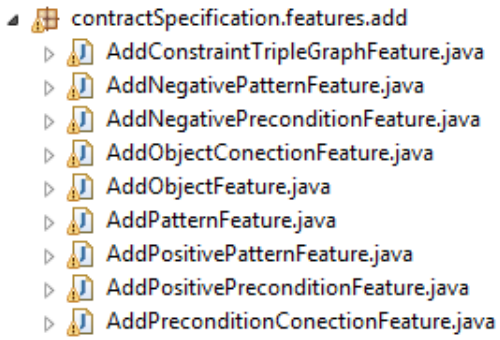


Figura 12. Clases Add del caso básico.

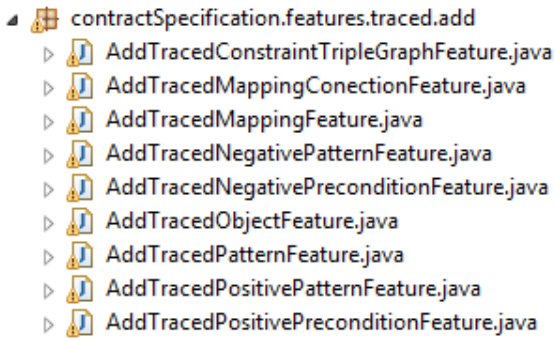


Figura 13. Clases Add del caso traced.

Es notorio que el propio nombre de la clase ya nos aporta conocimiento sobre qué es lo que estamos creando; por ejemplo, la clase *AddPositivePatternFeature*.java añade un patrón positivo. Además, *AddPatternFeature* es una clase de la que heredan tanto su versión positiva como su versión negativa y que las demás clases del caso básico son independientes. Por otro lado en el caso traced podemos observar una pequeña redundancia. Exceptuando los añadidos referentes a los objetos y mappings (éstos últimos heredan de los primeros ya que solo cambia el color de su rectángulo) heredan de sus correspondientes casos básicos y simplemente han sido extendidos con una funcionalidad adaptada al caso *traced*.

6. 2. 3. Move

Todas las figuras pueden ser movidas gracias a la funcionalidad básica que nos ofrece Graphiti por defecto. Aún así, hay otras que, por defecto, pueden ser movidas y en nuestro caso no deberían y también hay otras que parte de moverse de la manera básica deben contemplar otros aspectos.

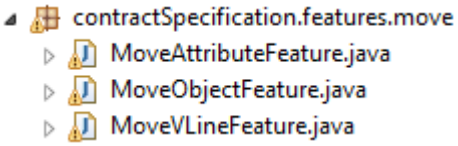


Figura 14. Clases Move del caso básico.

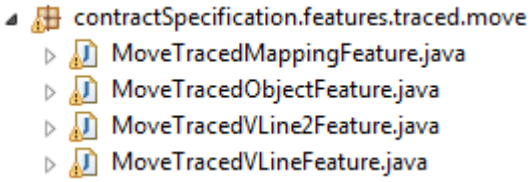


Figura 15. Clases Move del caso traced.

En el caso básico tenemos: unos atributos que no pueden ser movidos, unos objetos que no pueden sobrepasar el rectángulo en el que se encuentran ni la línea vertical que separa el origen del destino y esa misma línea vertical que no puede salir del rectángulo mencionado.

El caso *traced* es similar. Los mappings no pueden salir de la columna del centro, los objetos no pueden salirse del rectángulo ni sobrepasar las líneas verticales, y esas mismas no pueden cruzarse entre sí ni salir del rectángulo.

6. 2. 4. Layout

Sólo hay dos elementos en el diagrama que pueden ser modificados en cuanto a tamaño se refiere: los objetos y los patrones. No vamos a entrar en detalle puesto que no es muy relevante. Simplemente decir que en los patrones podemos encontrar líneas verticales y horizontales y en los objetos líneas horizontales y atributos. Simplemente estos componentes se alargan o encojen en función de lo que haga su contenedor. Además, se ha tenido en cuenta que si agrandamos un objeto de tal forma que desborde las dimensiones de su contenedor o la zona que hay reservada para él, su tamaño se ajuste correctamente y no se produzca ese desbordamiento.

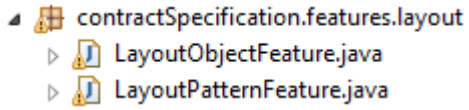


Figura 16. Clases Layout del caso básico.

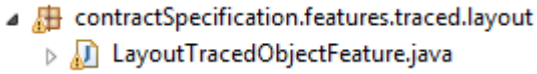


Figura 17. Clase Layout del caso traced.

6. 2. 5. Resize

Al igual que la funcionalidad de mover, el *resize* viene implementado por defecto. En nuestro caso hemos creado tres clases *resize* (las que vemos en la **Figura 18**) que impiden que tanto las líneas verticales que separan los grafos, como los atributos y como los grafos triples o patrones colapsados sean redimensionados.

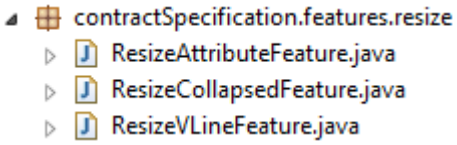


Figura 18. Clases Resize.

6. 2. 6. Delete

Vuelve a aparecer una *feature* que ya está implementada por defecto por Graphiti con una funcionalidad básica. Por norma general todas las figuras que usamos utilizan ese borrado por defecto ya que para borrarlas no es necesario realizar ninguna otra acción. Aún así, tenemos tres casos en los que ha sido necesario implementar nuestra propia *feature* de borrado. Estos casos coinciden con los elementos que podemos intuir de las siguientes clases:

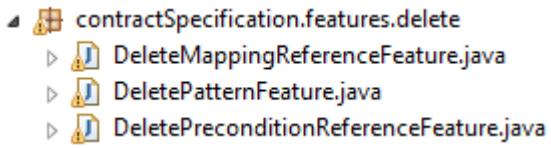


Figura 19. Clases Delete.

Cuando creábamos una precondición en un patrón, se creaba un grafo triple unido al patrón mediante una flecha. La flecha podía ser borrada y la precondición quedaba descolgada. La *feature delete* asociada al borrado de esa flecha hace que no se pueda borrar y, al mismo tiempo, la *feature* asociada al patrón hace que sus precondiciones sean eliminadas también (al eliminar un elemento de la conexión se elimina ésta también). Por otro lado, como las referencias de los mappings no tienen ningún objeto de dominio asociado a ellas, pasa lo siguiente: al eliminar esta referencia, no se elimina la relación. Si eliminamos la referencia que une el mapping con un objeto, ésta no será eliminada en el modelo de dominio. Por eso, esta *feature* hace eso mismo.

6. 2. 7. Actions

Tanto los patrones como los objetos tienen un menú contextual asociado en el cual se encuentran unos botones que ejecutan ciertas acciones. El encargado de ejecutar estas acciones al pulsar los botones es el *ToolBehaviourProvider* (dependiendo el botón que pulsemos, llama a un método u a otro). La funcionalidad reside en las clases del paquete de acciones. Éstas son las que vemos en la **Figura 20** y **Figura 21**.

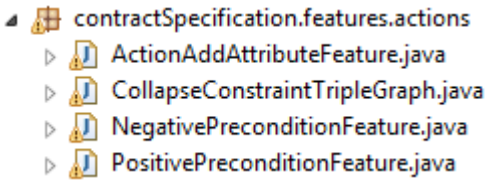


Figura 20. Clases de acción del caso básico.

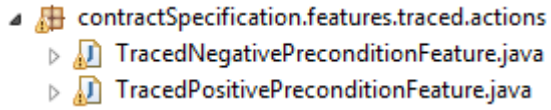


Figura 21. Clases de acción del caso traced.

Como se puede ver, tenemos la acción que crea un atributo en un objeto (la propia clase se encarga de crear la representación gráfica del atributo y hacer el *link* correspondiente con el objeto de dominio que también se crea), la acción que colapsa patrones y grafos triples (reduce/restablece el tamaño del elemento y lo etiqueta como colapsado/expandido) y la acción de crear precondiciones que aparte de crear el objeto de dominio correspondiente y crear la conexión entre la precondición y el patrón, utiliza la clase ya explicada de añadido de precondiciones (se ha explicado en la sección 6.2.2) para la representación gráfica de la figura. Las clases del caso *traced* heredan de las clases del caso básico adaptando la funcionalidad para el nuevo campo de cada precondición.

6. 2. 8. Properties

Ya se mencionó con anterioridad que algunos elementos del diagrama tenían asociada a ellos una hoja de propiedades a través de la cual se podía ver y modificar información sobre éstos. Como veremos a continuación, cada hoja de propiedades consta de una parte llamada *Section* y de otra llamada *Filter*. En la primera se estipulan los campos de la hoja ya sean campos de texto, desplegables, ... y en la segunda se hace un filtro para, a partir del elemento que recibe, saber qué hoja de propiedades debe de mostrar.

En la **Figura 22** vemos que los elementos que tienen una hoja de propiedades son los atributos, las conexiones entre objetos, el propio diagrama, los patrones y grafo triples y los objetos. En este mismo orden iremos explicando qué se puede ver en cada una de ellas: En los atributos podremos cambiar el nombre y el valor/variable de éstos. En las conexiones podremos editar su nombre. En la hoja del propio diagrama podremos cambiar los metamodelos cargados (e incluso descargar) en cada uno de los grafos. Podremos cambiar el nombre de los patrones (y grafos triples que como ya dijimos guardan una relación de igualdad), y añadir/eliminar condiciones de los atributos. Por último, en cada objeto podremos modificar su nombre y el tipo (mediante un campo de texto o un desplegable con los tipos que ofrezca el metamodelo cargado si lo hubiera).

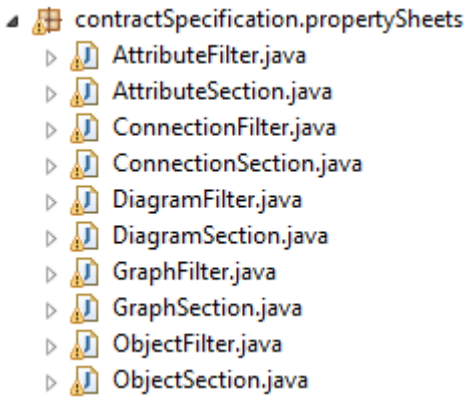


Figura 22. Clases para representar las hojas de propiedades.

Paralelamente, para que las hojas de propiedades se muestren correctamente es necesario modificar el archivo plugin.xml. Vemos un ejemplo de uno de los extractos (extrapolar para el resto de los casos) en la **Figura 23** donde además podemos ver cómo se especifican la clase *Section* y su clase *Filter*.

```
      <extension
      point="org.eclipse.ui.views.properties.tabbed.propertySections">
<!--     <propertySections contributorId="tfg.PropertyContributor"> -->
    <propertySections contributorId="ContractSpecification.PropertyContributor">
       <propertySection
             class="contractSpecification.propertySheets.DiagramSection"
             filter="contractSpecification.propertySheets.DiagramFilter"
             id="graphiti.main.tabSpecification.connection"
             tab="graphiti.main.tabSpecification">
       </propertySection>
    </propertySections>
   </extension>
```

Figura 23. Hoja de propiedades del diagrama en el archivo plugin.xml.

7. Resultados

Este apartado está dedicado a ilustrar el funcionamiento del proyecto e indagar un poco más en todas las posibilidades que éste brinda al usuario.

Primero nos centraremos en el funcionamiento básico de creación de figuras y conexiones y después mostraremos las hojas de propiedades de cada elemento y cómo éstas interactúan con el diagrama y el modelo de dominio.

7. 1. Diagrama

Lo primero de todo es que al crear un proyecto con Graphiti [6] siguiendo el template que hemos desarrollado nosotros (llamado *ContractSpecification*) antes de poder interactuar con él, necesitamos tomar una decisión frente al formulario que se nos aparece.

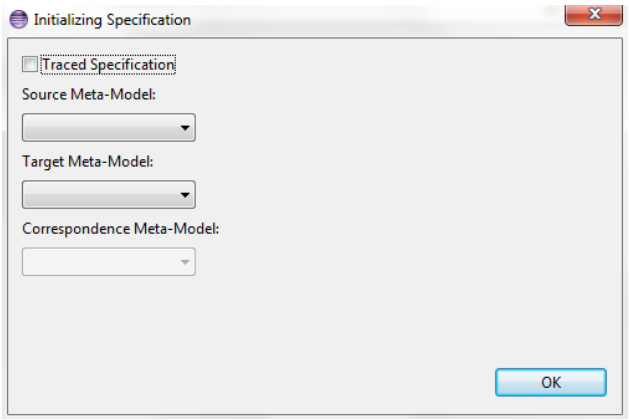


Figura 24. Formulario de inicio vacío.

Este formulario sirve para elegir si el diagrama va a constar de elementos *traced* (con su columna extra y todo lo demás que conlleva) y también qué metamodelos estarán cargados desde el principio. Como podéis ver, en la **Figura 24** el diagrama que se va a crear no es *traced* por lo que el campo *Correspondence Meta-Model* esta desactivado.

En la **Figura 25** vemos que la casilla de *traced* está activa por lo que el campo *Correspondence Meta-Model* está activado y se procedería a cargar el metamodelo "mutatorenviroment" en el grafo de correspondencia.

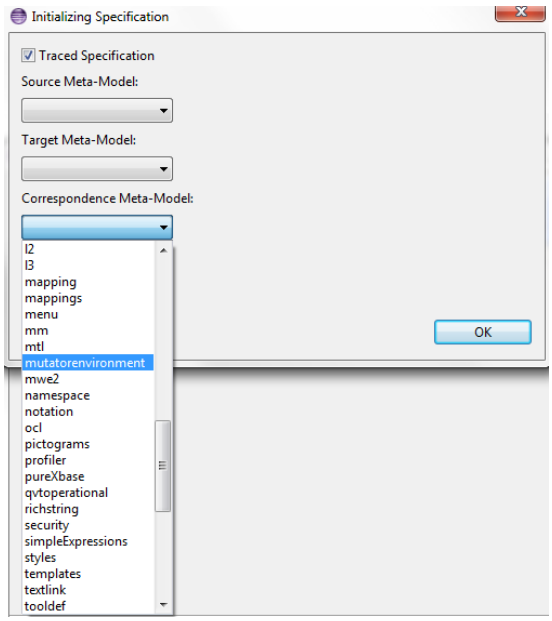


Figura 25. Formulario de inicio traced.

Ahora bien, dependiendo de qué tipo de diagrama hayamos elegido, el diagrama mostrará una paleta u otra. En el caso básico quedaría así:

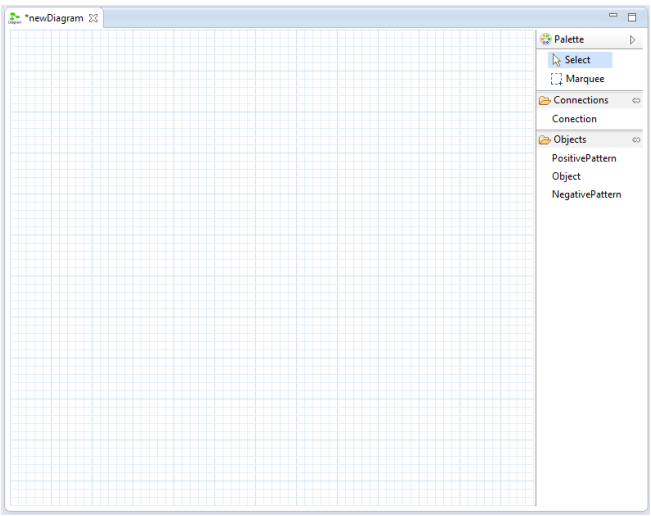


Figura 26. Diagrama básico.

El caso *traced* sería exactamente igual salvo que en su paleta estaría incluido también el elemento *Mapping*.

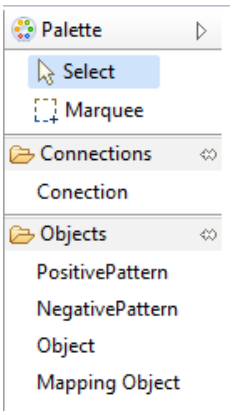


Figura 27. Paleta de diagrama traced.

La hoja de propiedades de la especificación está asociada al propio diagrama y podemos verla cuando hacemos *click* sobre éste. Su apariencia es la siguiente:

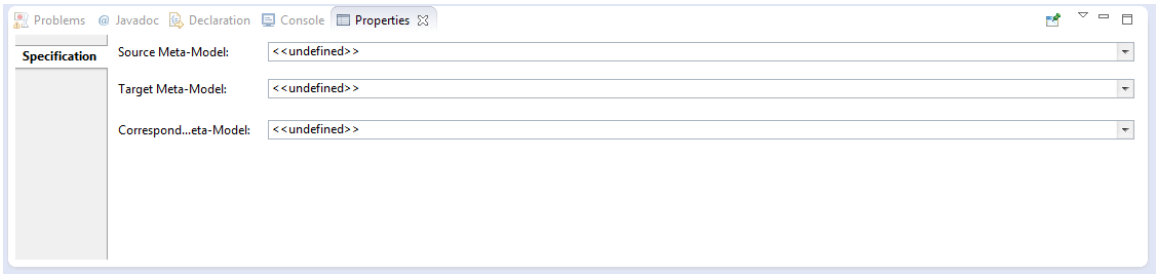


Figura 28. Hoja de propiedades de la Specification.

Sus campos representan los metamodelos cargados en los diferentes grafos. Cambiando el valor en la hoja de propiedades hará que los nuevos metamodelos se carguen donde corresponda. En el caso básico, el campo correspondiente con el metamodelo de correspondencia estaría deshabilitado.

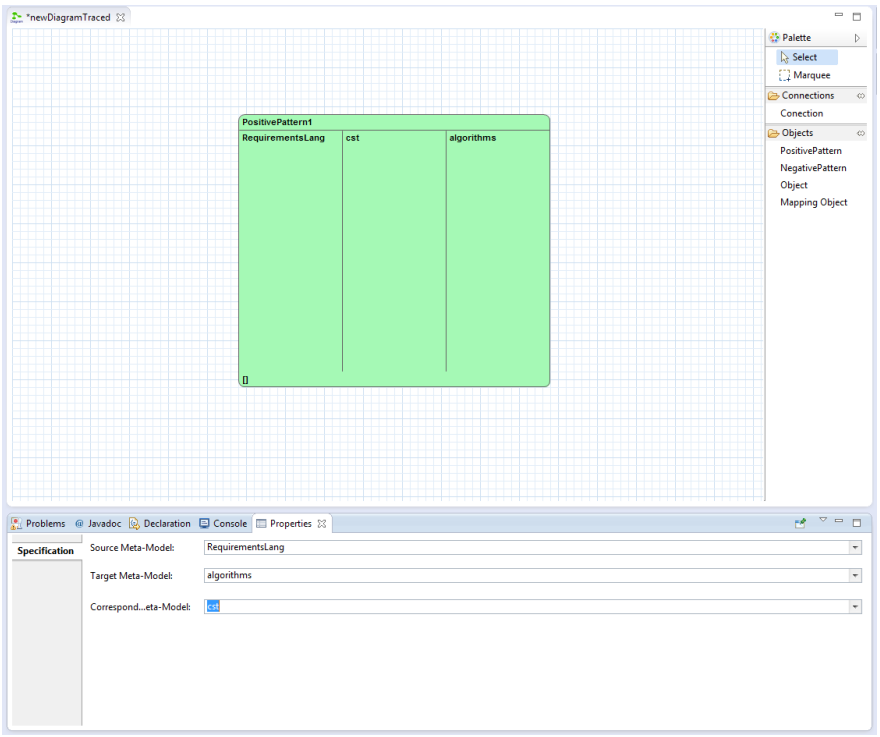


Figura 29. Metamodelos cargados.

7. 2. Patrones: Patrones Positivos y Patrones Negativos

En este apartado nos basaremos en el caso básico ya que el caso *traced* es exactamente igual salvo que los patrones tienen una columna central como bien se ilustró en la sección 4.3. Si clicamos sobre *PositivePattern* en la paleta, el editor nos ofrece la posibilidad de crear un patrón positivo en el diagrama. Además, aparece un formulario para que rellenemos el nombre del patrón que vamos a crear.

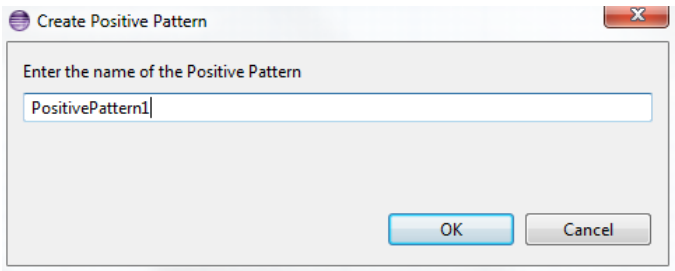


Figura 30. Formulario de creación de patrón.

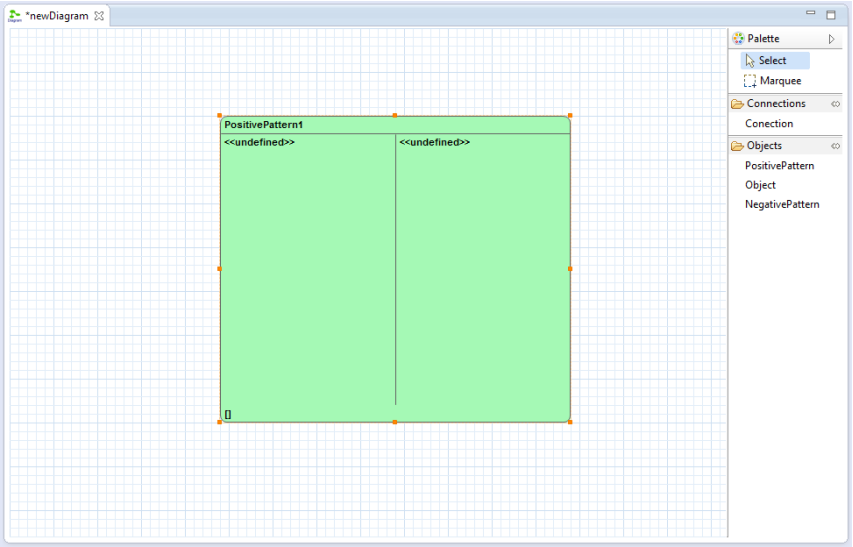


Figura 31. Patrón positivo creado en el diagrama.

Si intentamos crear ahora un patrón negativo dentro del patrón positivo que acabamos de crear, el editor no nos dejará finalizar la acción. Podemos observar esto en la siguiente figura fijándonos en el símbolo de prohibido que sustituye a la flecha del ratón:

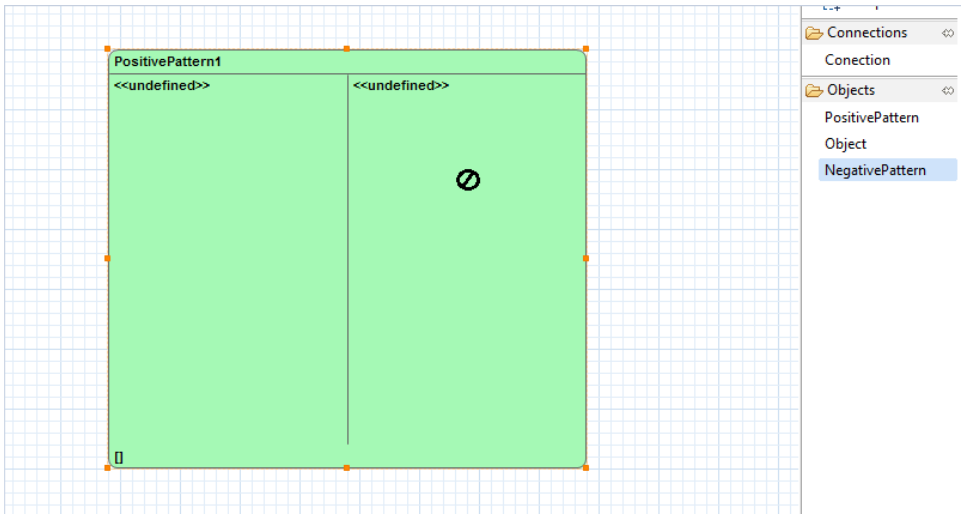


Figura 32. Prohibición de creación de patrón dentro de otro.

Además como ya hemos dicho, nos avisa cuando existen dos patrones con el mismo nombre:

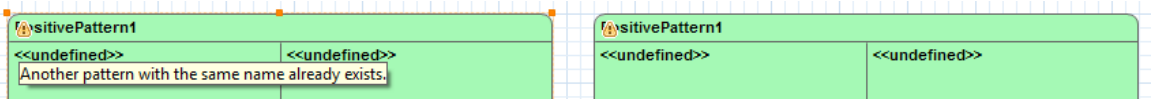


Figura 33. Warning por repetición de nombre en patrón.

Finalmente si creamos un patrón negativo en el diagrama, el resultado quedaría así:

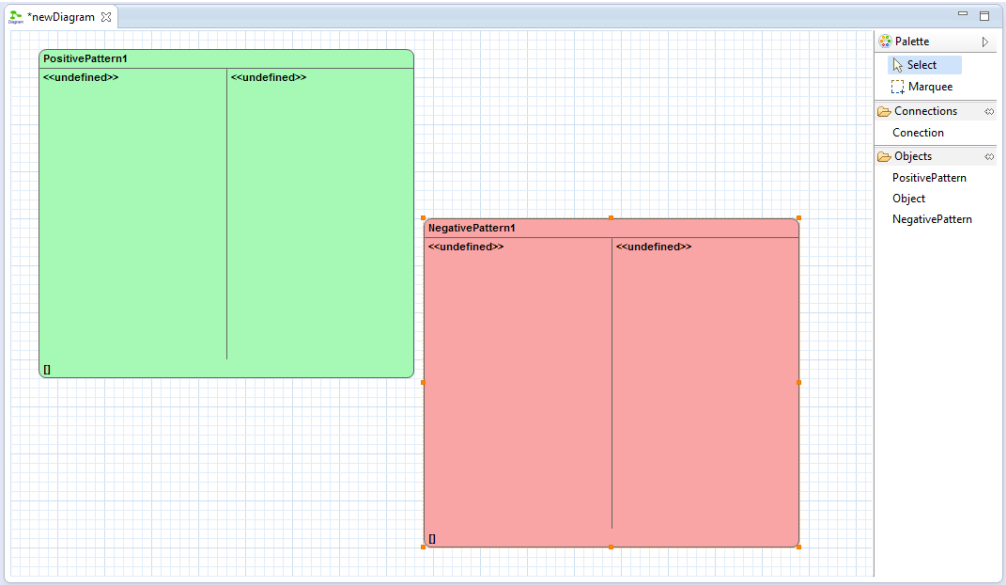


Figura 34. Patrón negativo creado en el diagrama junto a patrón positivo.

La hoja de propiedades asociada tanto a patrones como a precondiciones positivas y negativas es la misma y su apariencia es la siguiente:

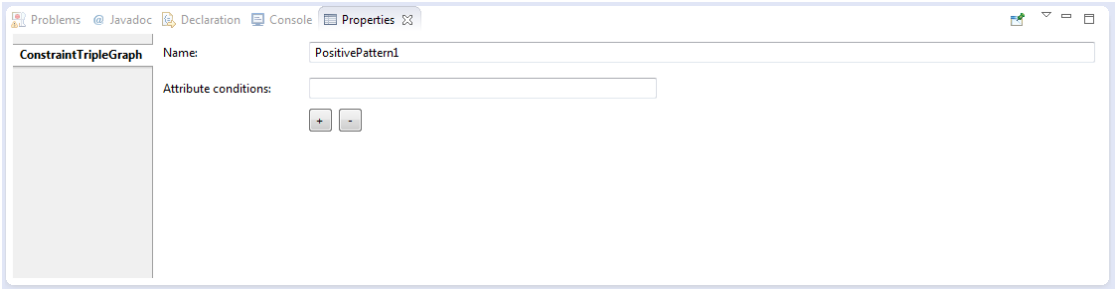


Figura 35. Hoja de propiedades de Patterns y ConstraintTripleGraph.

El campo llamado "*Attribute conditions*" sirve para añadir condiciones en forma de texto. El usuario escribe algo y lo añade con el botón + (también puede eliminarlo con el botón -). Cuando modificamos este campo de la hoja de propiedades, las precondiciones también se añaden a la figura, la cual tenemos seleccionada.

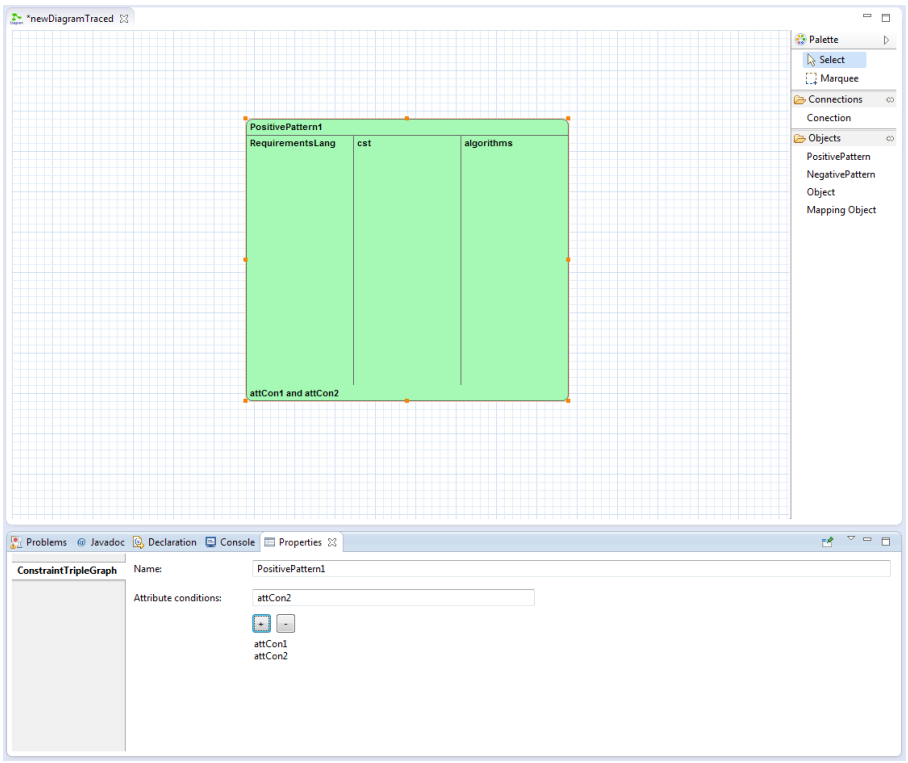


Figura 36. Attribute conditions modificadas.

7. 3. Objetos y Mappings

Antes que nada, si intentamos crear un objeto fuera de un patrón o un grafo triple aparecerá el mismo icono de prohibido que hemos mencionado antes. Recordar que podemos crear el objeto en cualquiera de los dos grafos (columnas) del patrón/grafo triple quedando de la siguiente manera:

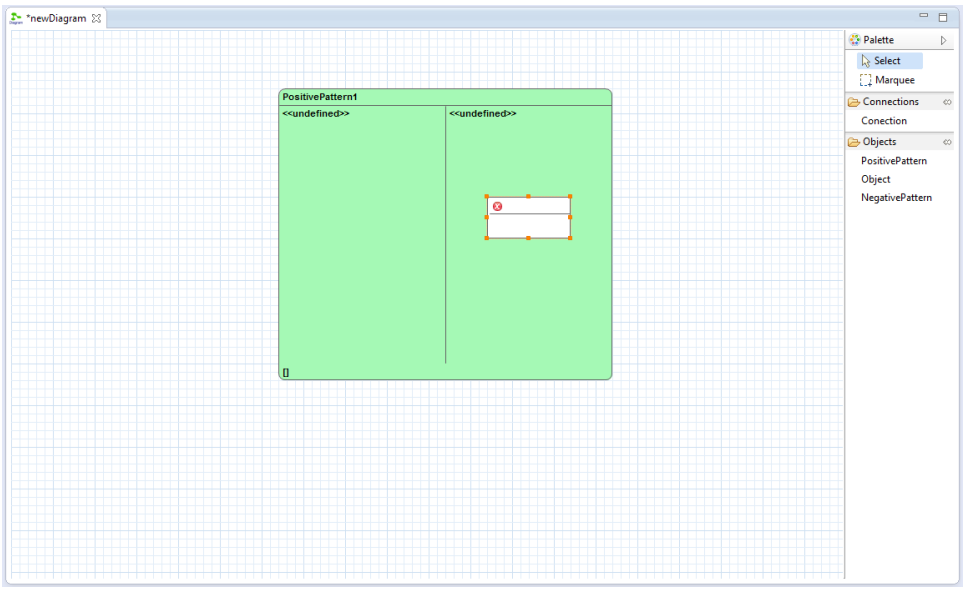


Figura 37. Object creado dentro de patrón positivo.

Con los mappings sucede lo mismo salvo que el color es diferente y que estos elementos sólo pueden ser creados en el grafo de correspondencia (la columna del medio). Si intentamos crearlos en el diagrama o en los otros dos grafos aparecerá el icono de prohibido. El resultado sería el siguiente:

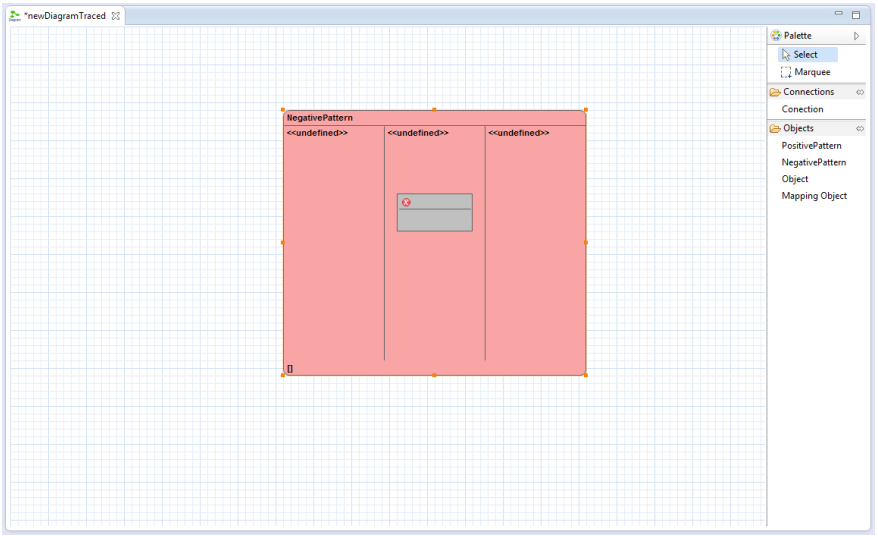


Figura 38. Mapping creado en patrón negativo.

Tanto en la **Figura 37** como en la **Figura 38** aparece un icono de error sobre los elementos creados. Esto es debido a que no tienen un nombre ni un tipo. Más adelante cuando se los demos a través de las hojas de propiedades, desaparecerán.

La hoja de propiedades asociada tanto a objetos como a mappings (un mapping también es un objeto) es la misma y su apariencia varía dependiendo si tenemos un metamodelo cargado en el grafo en donde se encuentra o no.

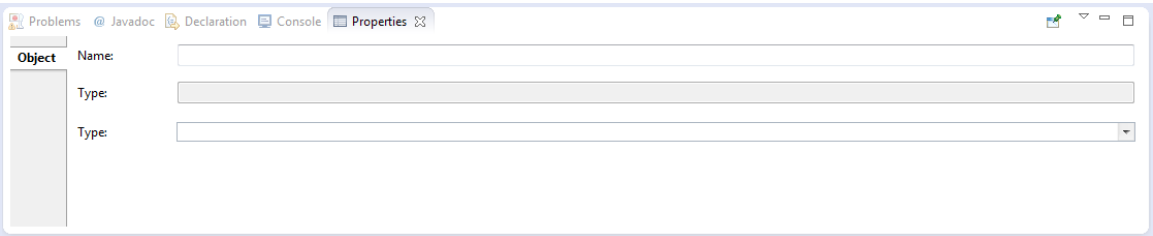


Figura 39. Hoja de propiedades de un objeto con un metamodelo cargado (desplegable).

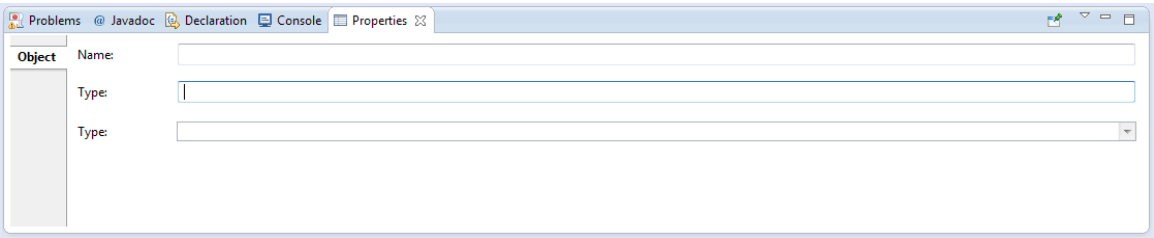


Figura 40. Hoja de propiedades de un objeto sin metamodelo cargado (campo de texto).

La única diferencia entre una y otra es que en una puedes escribir el tipo manualmente y en otra eliges de una lista de tipos que varía dependiendo del metamodelo que tengas cargado. Cuando modificamos la hoja, se modifica el objeto.

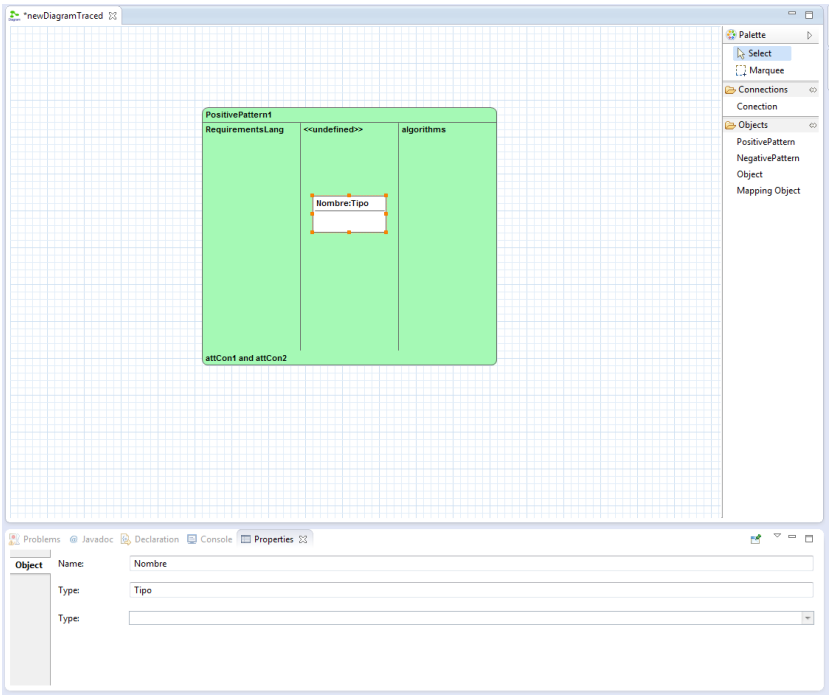


Figura 41. Campos del objeto modificados.

Los objetos al igual que los patrones no pueden tener el mismo nombre dentro de un mismo patrón:

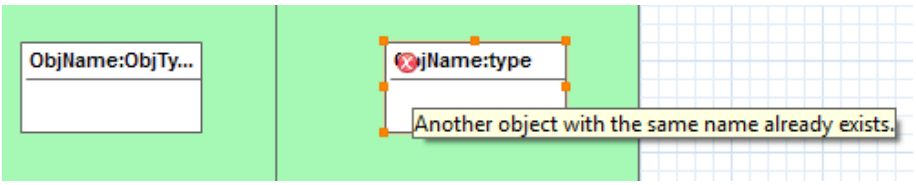


Figura 42. Objetos con el mismo nombre.

7. 4. Precondiciones Positivas y Precondiciones Negativas

Sobre un patrón creado podemos añadirle precondiciones positivas y negativas (tanto en el caso básico como en el caso *traced* por lo que sólo ilustraremos uno de los dos) haciendo *click* en el botón derecho sobre éste y eligiendo la opción adecuada del menú contextual.

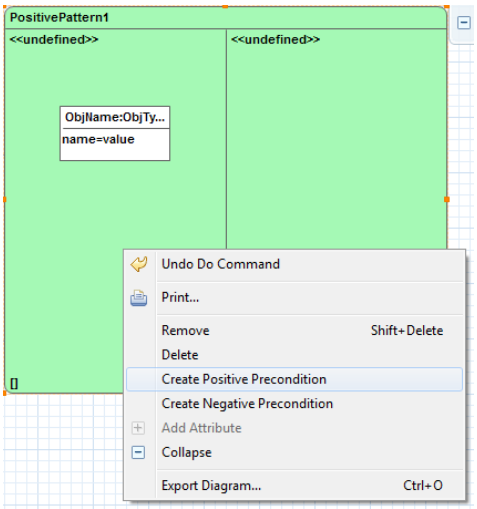


Figura 43. Menú contextual de los patrones y grafos triples.

Una vez creada la precondición positiva el resultado es el que viene a continuación (tras la aparición del formulario que nos pide el nombre que le queremos dar):

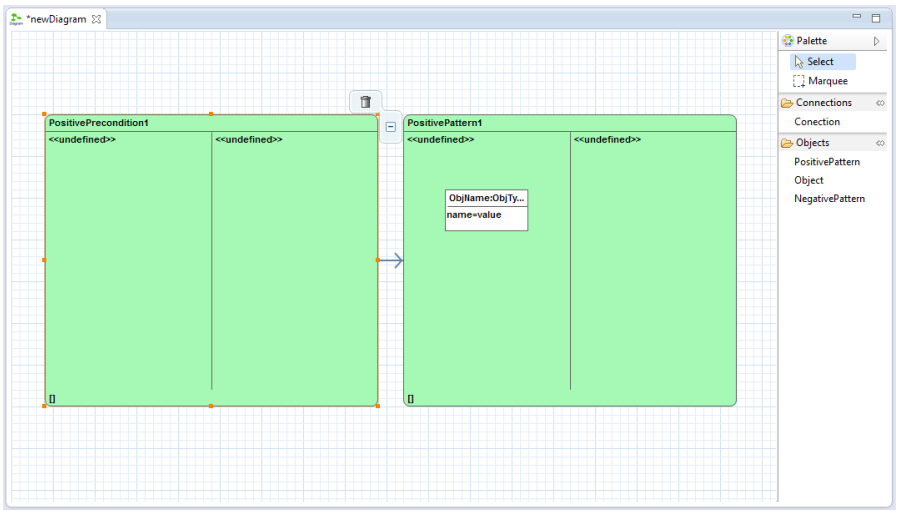


Figura 44. Precondición positiva creada.

Cómo sólo podemos tener una precondición positiva, si intentamos crear otra aparecerá un mensaje avisándonos de que no podemos tener más.

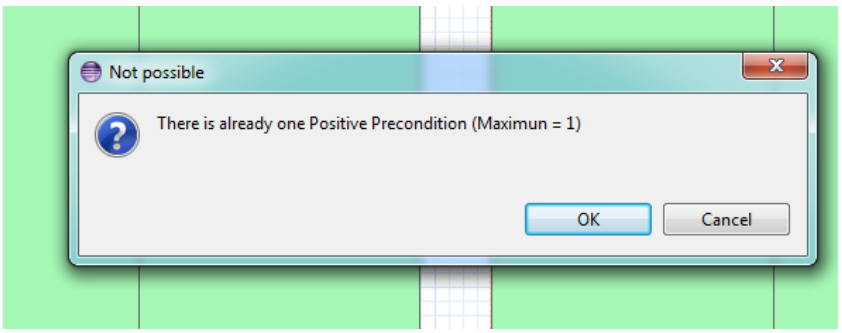


Figura 45. Mensaje de aviso sobre exceso de precondiciones.

En cuanto a lo referente con las precondiciones negativas el caso es el mismo salvo que su color es rojo y que además podemos tener un número ilimitado de ellas. Debemos recordar que las precondiciones actúan de la misma forma que los patrones en relación a los objetos. Éstos y los mappings también pueden ser añadidos de igual manera. Cuando un patrón se borra, se borran todas sus precondiciones.

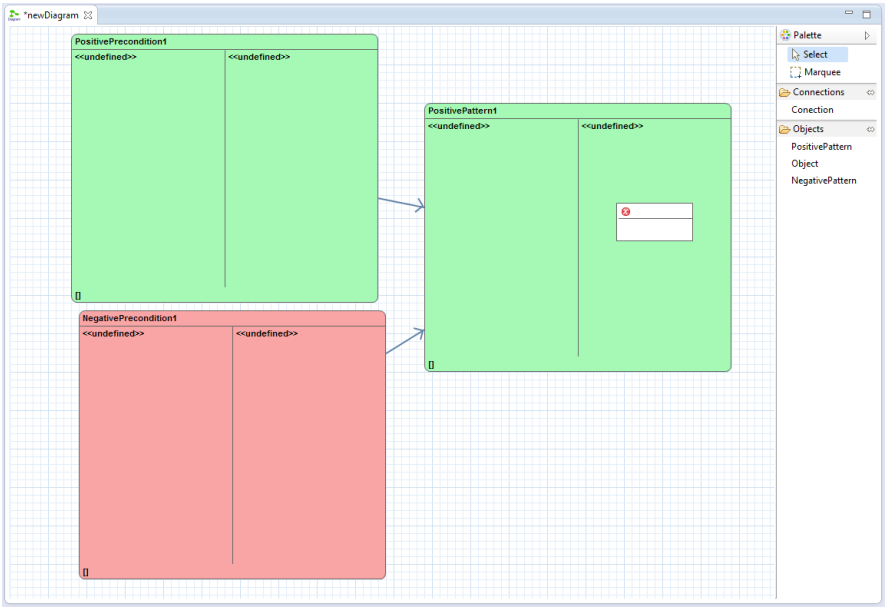


Figura 46. Precondición negativa creada junto a precondición positiva.

7. 5. Atributos y Referencias

Los atributos pueden ser creados mediante el menú contextual que aparece al clicar con el botón derecho sobre los objetos o sobre el menú contextual que aparece cuando situamos el ratón sobre ellos (clicando en el icono con un símbolo +). Una vez creado aparece vacío y con un icono de error ya que no tiene ningún campo completo. Cuando hablemos de las hojas de propiedades estos iconos desaparecerán.

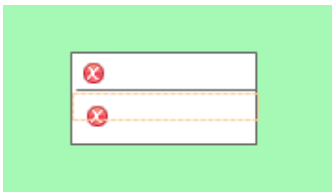


Figura 47. Atributo añadido al objeto.

La hoja de propiedades de los atributos sólo tiene dos campos: *name* y *value*. Sin embargo la clase *Attribute* tiene 3 atributos. La clave está en que si el campo *value* lo rellenamos entre comillas, este valor se guardará en un atributo mientras que si lo hacemos sin comillas lo hará en el otro. En el primer caso el valor del atributo se interpretará como un valor constante (ej. “value”), mientras que en el segundo caso se interpretará como una variable (ej. value) y su valor vendrá determinado por las condiciones sobre atributos definidas en la parte inferior del patrón.

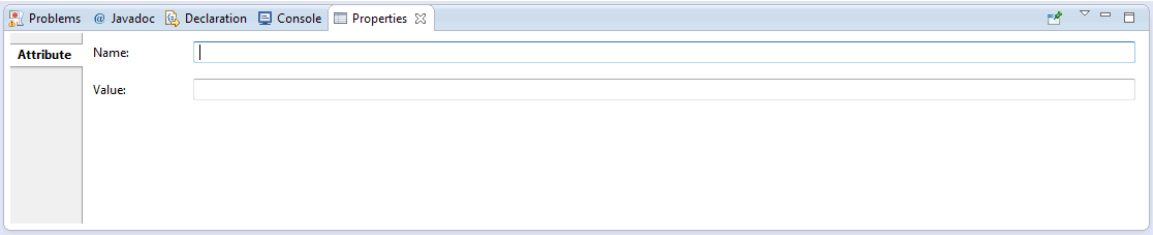


Figura 48. Hoja de propiedades de un atributo.

Si observamos la siguiente imagen podremos ver cómo al modificar la hoja de propiedades se actualizan los valores de la figura que representa al atributo también.

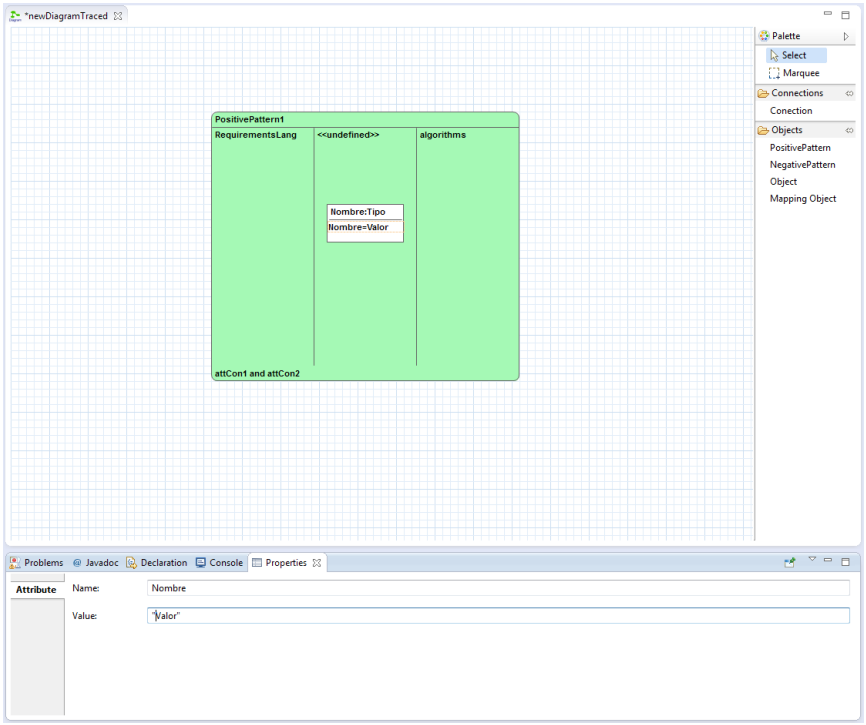


Figura 49. Campos del atributo modificados.

En el caso básico, las referencias se realizan entre objetos dentro del mismo grafo. En el caso *traced* es lo mismo salvo que los mappings situados en el centro pueden referenciar a objetos del mismo grafo o de los otros dos. En cualquier otro caso el editor nos muestra un icono de prohibido. Recordamos que los mappings sólo pueden relacionar a un objeto de los grafos origen y destino al mismo tiempo y que las referencias a sí mismo están permitidas. Las referencias pueden ser creadas mediante la paleta o mediante el menú contextual que aparece al situar el ratón encima del objeto (un icono en forma de flecha). A continuación veremos varias imágenes en las que exponemos todos los casos posibles de referenciación:

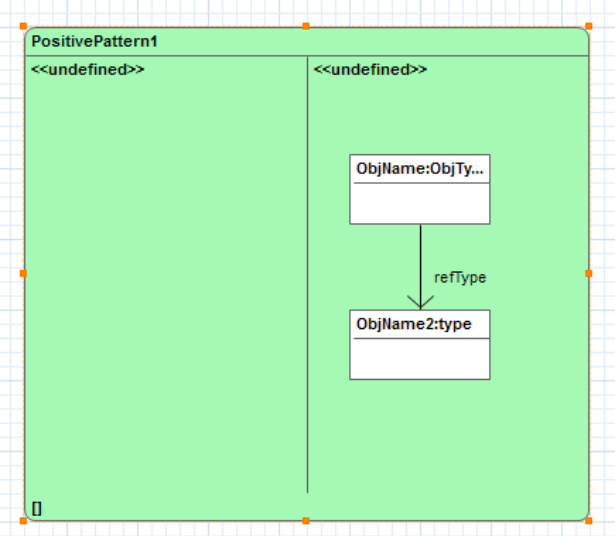


Figura 50. Objeto referenciando en el grafo de destino.

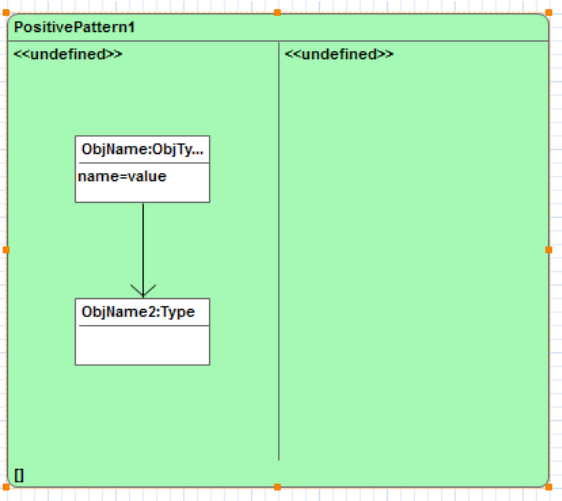


Figura 51. Objeto referenciando en el grafo de origen. Referencia sin nombre.

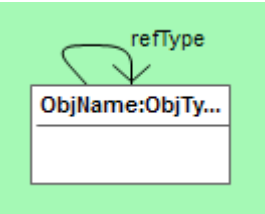


Figura 52. Objeto referenciándose a sí mismo.

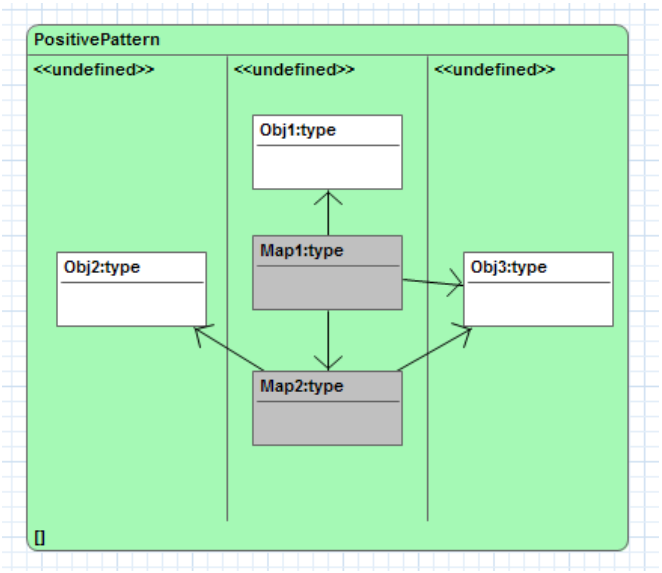


Figura 53. Situación completa de referenciación con Mappings.

No hay que confundir estas referencias con las referencias que salen desde un mapping a un objeto. Éstas últimas no tienen ningún objeto de dominio asociado por lo que no tienen ninguna hoja de propiedades.

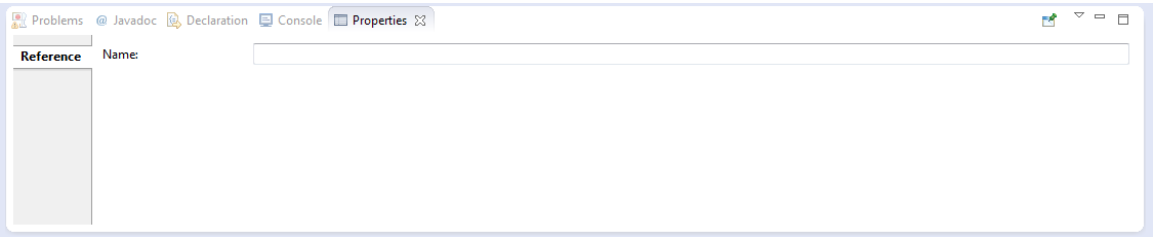


Figura 54. Hoja de propiedades de una referencia.

En relación a las referencias hay varios aspectos a contemplar. Cuando no hay ningún metamodelo cargado en el grafo en el que nos encontramos, pueden tener cualquier nombre sin problema alguno. El problema viene cuando hay un metamodelo cargado. El nombre de la referencia tiene que existir en el objeto del que parte (la clase del metamodelo que representa ese objeto tiene que tener una referencia llamada así) y además el objeto al que apunta la referencia tiene que coincidir con el tipo de objetos a los que puede apuntar esa referencia. A continuación veremos dos ejemplos de errores que se pueden cometer (errores que el propio editor nos notifica con su icono correspondiente) y el caso válido.

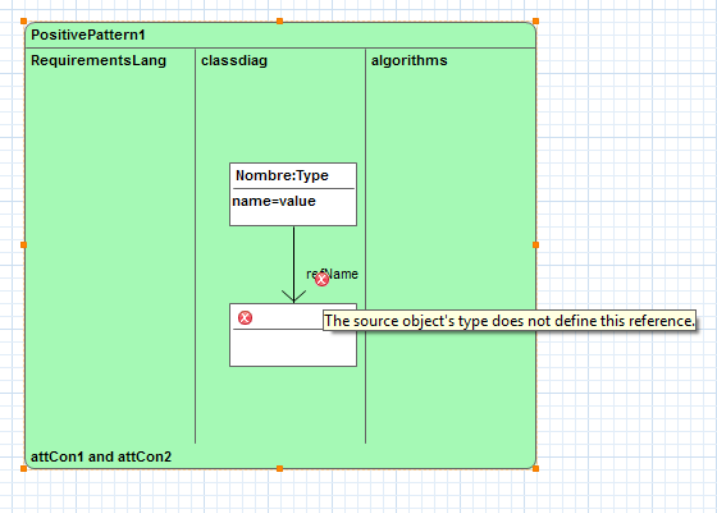


Figura 55. El objeto no tiene una referencia llamada “refName”.

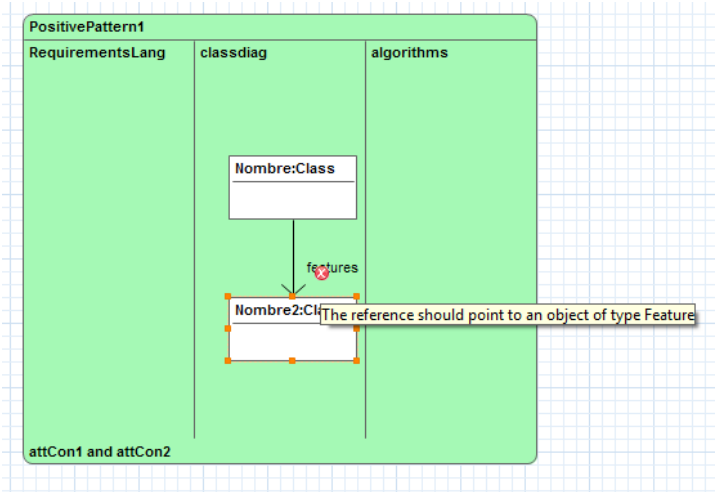


Figura 56. La referencia “features” no apunta a un objeto “Class”.

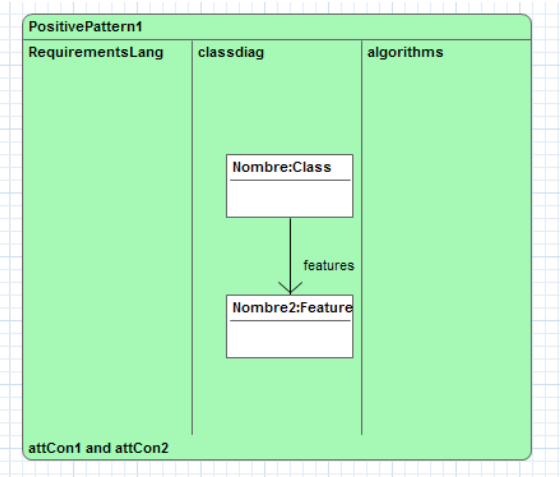


Figura 57. El objeto “Class” define una “Feature” en features. Caso válido.

8. Conclusiones y trabajo futuro

Se puede decir que el documento ha llegado a su fin. Aún así, todavía hay cosas que comentar sobre el proyecto. En esta sección hablaremos sobre qué conclusiones hemos podido sacar durante el desarrollo y también sobre qué hacer una vez finalizada esta primera versión del proyecto, qué trabajo futuro se realizará o puede ser realizado.

8. 1. Conclusiones

Puesto que Graphiti [6] era desconocido, ha sido necesario pasar por una fase de aprendizaje lo cual, aunque resulte más trabajoso para el desarrollo, hace que termines comprendiendo mejor cómo funciona el *framework* que estás aprendiendo. El hecho de "pelearte" con una herramienta hace que aprendas de los errores cometidos (siempre se cometen errores al principio).

Podemos decir que Graphiti [6] es un *framework* muy útil para crear editores gráficos en un periodo corto de tiempo (más corto cada vez según el desarrollador esté más familiarizado con él) y de una calidad más que aceptable. Al estar integrado en Eclipse [5], uno de los entornos de desarrollo más reconocidos en el mundo de la informática, hace que muchos usuarios puedan disfrutar de él y por ende, que la herramienta siga en desarrollo mejorando y ampliando su funcionalidad.

La gran mayoría de editores que pueden ser realizados mediante Graphiti [6] son muy similares. Constan de casi las mismas funcionalidades y la estructura es muy similar. Por ello, siguiendo el tutorial que aporta la propia herramienta facilitó y aceleró el desarrollo de este proyecto.

Para terminar, decir que el resultado final es bastante sólido y cumple con todo lo que se planteó en un principio. No ha sido necesario desechar ninguna funcionalidad por no ser posible su implementación o por cuestión de tiempo. Su funcionalidad puede ser extendida fácilmente y de eso hablaremos en la siguiente sección.

8. 2. Trabajo futuro

Esta sección deja el futuro desarrollo del proyecto muy abierto. Como bien hemos dicho hemos cumplido con todo lo que había que implementar por lo que se podría decir que no hay nada más que hacer (el proyecto tampoco tiene una funcionalidad que abarque muchos aspectos).

Algo que si se podría resaltar, es que al ser un editor gráfico, puede que haya numerosos usuarios a los que no les convenza la apariencia de éste. Puede que las figuras o los colores no convenzan por lo que se puede valorar el hecho de la existencia de un *wizard* de personalización para que el usuario configure todo a su gusto.

Por último, ya comentamos la existencia de un editor textual que definía contratos para transformaciones de modelos. Éste además incluye un generador de scripts que permite validar una transformación respecto al contrato por lo que si de alguna manera se integraran ambos editores (este editor textual y el editor gráfico que hemos desarrollado) sería posible realizar una validación gráfica.

9. Bibliografía

[1] Wikipedia, MDE. Available at : http://en.wikipedia.org/wiki/Model-driven_engineering [Accedido noviembre 21, 2013].

[2] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev. "ATL: a model transformation tool".2008. Science of Computer Programming 72:31-39 (Elsevier). See also http://www.eclipse.org/atl/

[3] Wikipedia, QVT1.1. Available at: http://www.omg.org/spec/QVT/ [Accedido noviembre 21, 2013].

[4] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige and Osmar Marchi dos Santos. "Engineering model transformations with transML". 2013. Software and Systems Modeling (Springer). Volume 12, Issue 3, pp.: 555-577.

[5] Eclipse. Mainpage. Available at: http://www.eclipse.org/ [Accedido noviembre 21, 2013].

[6] Graphiti, a Graphical Tooling Infraestructure. Available at: http://www.eclipse.org/graphiti/ [Accedido noviembre 21, 2013].

[7] Eugenia. Available at: http://www.eclipse.org/epsilon/doc/eugenia/ [Accedido noviembre 21, 2013].

[8] GMF. Graphical Modeling Project (GMP). Available at: http://www.eclipse.org/modeling/gmp/ [Accedido noviembre 21, 2013].

[9] Overview of Domain-Specific Language Tools. Available at: http://msdn.microsoft.com/en-us/library/bb126327.aspx [Accedido noviembre 21, 2013].

[10] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks."EMF: Eclipse Modeling Framework", 2nd Edition. Addison-Wesley Professional, 2008. Available at: http://www.eclipse.org/modeling/emf/

[11] GEF (Graphical Editing Framework). Available at: http://www.eclipse.org/gef/ [Accedido noviembre 21, 2013].

[12] Graphiti developer guide. Available at http://help.eclipse.org/kepler/index.jsp?nav=%2F25 [Accedido noviembre 21, 2013].